A Unified Framework for Constraint-based Modeling

Thesis by
Devendra Kalra

California Institute of Technology
Pasadena, California

1990

A Unified Framework for Constraint-based Modeling

Thesis by
Devendra Kalra

In Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy

California Institute of Technology
Pasadena, California

1990
(Submitted May 31, 1990)

Acknowledgments

My foremost thanks go to Al Barr, my advisor, for his advice on my research, for his friendship and for his trust. Al has created a wonderful intellectual and physical environment at the Graphics Lab at Caltech and it was a great learning experience working in Al's lab. He has provided excellent advice, flexibility and moral support throughout my graduate career.

I would also like to thank Ronen Barzel for his extensive cooperation in software design and implementation, for his ideas and discussions, and for always being a very pleasant person to work with. Thanks go to Jim Kajiya for very useful discussions. Thanks go to Joel Burdick, Mani Chandy, Jim Kajiya and Steve Taylor for being on my committee and for their valuable suggestions.

Amar Gandhi helped with modeling for some of the simulation examples presented in this thesis. My thanks also go to Tim Kay, Dave Kirk, David Laidlaw, Mike Newton and John Snyder for their help and friendship.

A special thanks to Carolyn Collins, group mother and secretary, for working so hard for all of us. Carolyn has been amazingly resourceful and supportive under extreme conditions of temperature and pressure!

Finally, big thanks to my parents, Sohan Lal and Suman Lata Kalra, for inspiring, encouraging and supporting the need for a good education.

All the software described in this thesis was developed on HP9000/835 machines donated very generously by Hewlett Packard to the Caltech Computer Graphics Group. The research described in this thesis was supported in part by NSF under Cooperative Agreement No. CCR-8809615.

Abstract

Constraint-based modeling techniques are emerging as an effective computer graphics approach for modeling and designing objects and their behaviors.

In this thesis, computer graphics constraint techniques are unified into a single conceptual framework. The central themes of the thesis are methods to partition an arbitrary constraint problem in different domains and at different levels, and to provide a language and computational environment for modeling with constraints. Using partitioning and composition schemes, complex simulations can be built hierarchically from simpler simulations by "plugging" together separate modules. Fundamental and basic structures are designed and implemented to provide an "Assembly Language" for simulation systems. These structures are put together through a collection of interfaces, much like multiple languages that use the same assembler on a computer.

We use strategies called *refinement* and *partitioning* to integrate seemingly disparate constraint techniques. We present *Temporal Sequencing* as an approach to design complex time behaviors of simulation systems.

Refinement is a top-down approach of transforming high level representations of a constraint modeling problem into representations that are closer to the basic solution mechanisms available in the constraint environment, such as numerical solution methods. Partitioning is the decomposition of one constraint problem into multiple simpler constraint problems that are then studied separately. Temporal Sequencing is a methodology to design the time behavior of a simulation system by composing time behaviors of the system over subintervals of time.

Using the above partitioning schemes for the solution and specification of a general constraint problem, we create a unified constraint environment with the capability to both solve constraint problem instances and to create specialized constraint systems. New methods of constraint specification and solution can be added into the same constraint framework as new methods are developed.

Based on the above approach, a modeling system called "Our Constraint Environment"(OCE) has been implemented. A programming language as an extension to C++ has been designed to provide an interface to OCE. The language provides the constructs for the partitioning schemes discussed above. Simulations created using OCE have shown the efficacy of our design approach.

Contents

List of Figures

Part I

Introduction

Chapter 1

Introduction

1.1 Background

The process of creating images on a computer, can be broadly subdivided into two main activities, **modeling** [1] and **rendering**.

Modeling refers to the creation of mathematical models that represent objects and collections of objects. This involves creation of the geometry of individual objects, organization of objects to create a scene or in the case of **computer animation**, the time evolution of objects and their motion.

Rendering refers to the conversion of modeling data into an image. The input to the rendering process consists of mathematical descriptions of geometry and how light is to interact with the geometry. The output of the rendering process is a two-dimensional image composed of an array of discrete elements called **pixels**, each assigned a color. Computer animation is created by rendering a series of computer images and displaying the images in a sequence.

1.1.1 Current State of Modeling in Computer Graphics

Modeling in computer graphics may be roughly subdivided into three categories: **kinematic modeling**, **physically-based modeling** and **constraint-based modeling**

Kinematic Modeling

Kinematic modeling refers to "physicsless" modeling performed through mathematical structures such as functions or numbers. This includes the simple surfaces found in most computer modeled scenes: polygons, spheres and **parametric surfaces**. Objects and scenes are typically defined by directly specifying numbers to position

[1] Words found in the glossary are presented in **boldface font** when they are first used in the thesis.

and orient the objects (as well as for the parameters of the objects). Some of the earliest and still commonly used techniques to generate objects are curve-editors and procedural object generators with organizations of objects created as **transformation hierarchies** [FOLEY and VANDAM 82]. Similarly, in kinematic modeling, simple trajectories or **scripts** are used to control computer animation. Motion is specified, among other methods, by interpolating between **key-frames** or by specifying cubic parametric curves as paths for objects to move along ([STERN 83] [O'DONNEL and OLSON 81]).

Physically-based Modeling

Physically-based modeling refers to modeling objects while taking into account the underlying physics, usually Newtonian physics. The motion of rigid bodies may be modeled after Newton's equations of rigid body motion, the behavior of flexible bodies may use concepts of elasticity and plasticity theory, and collisions may be treated using momentum methods. Physically-based modeling provides a high degree of realism in the motion and shape of models with specification of much less detail as compared to kinematic modeling, but at the cost of additional computational effort. Without constraints, however, physical models are difficult to control, other than simulating straight forward initial value problems. For example, given the state of a rigid body and forces acting on it, the subsequent motion is computed from the current state.

Constraint-based Modeling

In constraint-based modeling, desired behaviors of simulated objects are expressed in terms of *constraints* among objects, or as *goals* that the objects must reach. It is desirable that the computer be able to translate the specifications of relationships and goals into the required numerical representation. Human beings seem to think more easily in terms of relationships between objects and do not seem to naturally think in terms of numeric values of parameters. We might wish to move an object between two points to optimize a criterion such as energy consumption. In another instance, our goal might be to put an object on a table. In this case, the desired relationship is "object **on** the table." We would usually prefer not to have to specify the exact distances and angles in advance that the object needs to be moved through so that it is on the table.

[BARR 88] uses the term *teleological* (from Greek word *telos* meaning end or goal) modeling for such an approach.

A Need to Unify Modeling Approaches

To design or specify complex systems of objects, we would like to use many different approaches to solve different sub-parts of the design and specification problem. Until this thesis, there did not exist a framework that unified the various constraint-based

approaches into an integrated modeling environment. We believe that creating an environment that integrates constraint-based, physically-based and kinematic modeling techniques will result in the capability of modeling complex systems. As part of the constraint unification, we will produce an **assembly language** for simulations so that the basic structures in this simulation assembly language can be put together to create versatile solutions, much like multiple high level languages on a computer can use the same assembler.

1.2 The Unification of Constraint Modeling

Given our need to unify different constraint-modeling approaches in the same framework, we need to answer two questions more precisely.

1. **What to unify?** We need to identify an extensible basis of modeling approaches.
2. **How to unify?** We need to create ways to make the techniques work together.

The thesis concentrates on solving the above problems.

1.2.1 Approaches for Constraint-based Modeling

We have chosen the following as an extensible collection of constraint-based modeling approaches[2].

- Inverse kinematics
- Inverse dynamics
- Constrained optimization
- Calculus of Variations
- Simulated annealing
- Hamiltonian (and Lagrangian) physics
- Differential-Algebraic Equations

The above constraint approaches have been developed for isolated types of constraint problems. Most of these approaches have evolved independently and were not created so that they would directly work together. Most implementations of constraint approaches have been quite specific, without the ability to interface smoothly with constraint approaches beyond the central approach in the implementation. At first glance, the approaches may seem quite different from each other. For example, the approaches of inverse dynamics, constrained optimization and Lagrangian physics have typically been unrelated.

[2]These approaches are described in appendices A and B.

1.2.2 Unifying Constraint-based approaches

As described in the previous section (1.2.1), we have chosen to unify a diverse collection of approaches. To create a strategy to unify these approaches, we use the following three steps:

(Step A) Create a set of primitive underlying elements for constraint-based modeling methods

(Step B) Identify the ways in which different elements interact with each other

(Step C) Create a computational formalism of the underlying elements that lets us use these techniques together

The above steps are independent of any specific approach to constraints and provide the design of a framework in which not only existing techniques can work together but future techniques can also be assimilated as they are developed.

We have chosen to define objects and constraints as the basic elements of constraint-based modeling. A **simulation entity**, a model of a system of interacting objects that we are interested in simulating, can be constructed out of the basic elements.

We have identified three general schemes to use in the solution of constraint problems. They are *refinement*, *partitioning*, and *temporal sequencing*. Refinement is a top-down approach of transforming "high" level representations of constraint problems into representations that are closer to basic solution mechanisms in the environment, such as numerical solution methods. Partitioning is the decomposition of one system of objects into multiple simpler systems that can be studied separately. Temporal Sequencing is a methodology to design the time behavior of a system by composing time behavior of the system over subintervals.

Based on this approach, we have implemented a modeling environment called OCE *(Our Constraint Environment)*. A programming language as an extension to C++ has been designed to provide an interface to OCE. The language provides some specialized constructs useful in writing simulation programs.

1.3 Brief History of Computer Graphics Modeling

Most of the early modeling work in computer graphics was based on kinematic modeling. Shapes of objects were designed using polygons and line segments or parametric curves and surfaces [BARTELS, BEATTY and BARSKY 83]. Collections of objects were organized using transformation hierarchies [FOLEY and VANDAM 82]. A number of kinematic motion control environments have also been reported that use various

techniques like scripts, key-frame interpolation and parametric curves to specify trajectories ([O'DONNEL and OLSON 81] [REYNOLDS 82] [STERN 83] [ZELTZER 84]).

Work in physically-based modeling started with the application of equations of rigid body motion to model computer graphics objects. [ISAACS & COHEN 87] [MOORE and WILHELMS 88] [HAHN 88] and [BARAFF 89] are examples of such work. [ARMSTRONG and GREEN 85] [WILHELMS 87] and [MILLER 88] have used dynamics for simulation of articulated bodies such as a human skeleton and worms. Simulation of elastic and plastic behavior of objects was presented in [TERZOUPOULOS et al 87] and [PLATT and BARR 88].

Isolated constraint-based modeling techniques, both based on kinematics and physically-based, have also appeared in the literature from time to time.

Sketchpad [SUTHERLAND 63] was a pioneering effort in both interactive computer graphics and in using constraints in computer graphics. Sketchpad was a two-dimensional graphical editor with lines and circular arcs as primitives. Constraints were specified between objects or parts of objects. Each constraint generated an error, a scalar, which was zero when the constraint was satisfied. These errors were first reduced using propagation of degrees of freedom and if this failed, a numerical iterative procedure was used.

Thinglab [BORNING 79], was written as an extension to the Smalltalk Language [GOLDBERG and ROBSON 83]. Thinglab was designed as a simulation laboratory to provide "an environment for constructing dynamic models of experiments in geometry and physics." A constraint was specified as a *rule* and a set of Smalltalk methods that can be invoked to satisfy the constraints. Thinglab used the rule to create a procedural test to check if the constraint was satisfied. Thinglab used simple constraint satisfaction techniques.

TEX [KNUTH 84] is a program that typesets text. It reduces a "penalty," a measure of deviation from a "good" state, to compute typesetting parameters such as amounts of space between words and paragraphs and where to start new lines and pages.

[BARZEL and BARR 88] uses an inverse-dynamics approach based on the Newton equations of rigid body motion. The individual elements in the environment described are rigid bodies. Constraint forces are computed that together with external forces cause the rigid bodies to meet the constraints specified on them. [ISAACS & COHEN 87] uses inverse-dynamics techniques in combination with kinematic constraints.

[WITKIN, FLEISCHER & BARR 87] and [PLATT 89] present numerical techniques based on optimization methods for **energy functions** to solve constraints on flexible elastic and plastic bodies. Bodies react to external forces and satisfy constraints such as impenetrability. [WITKIN and KASS 88] presents a technique to compute paths as functions over time by optimizing functionals of paths.

1.4 Benefits of a Unified Approach to Constraint-based Modeling

There are many reasons to develop a unified approach to constraint-based modeling that allows us to combine techniques together and helps us develop new ones.

1. *Utility:* Constraint-based modeling provides a useful technique to design objects and collections of objects. The specification of objects in terms of desired behaviors and goals provides an intuitive design methodology.

2. *Use of solution techniques appropriate to the natural structure of a problem:* The nature of complex problems usually suggests partitioning methods for decomposition of the problem into subproblems. The subproblems may require different techniques for their solution. To take advantage of the natural structure in constraint problems, we need the ability to solve each subproblem with techniques best suited for it. To be able to compose an overall solution, we also need that different techniques work in a complimentary manner. With the availability of a number of solution techniques that can be used in a complimentary way, we get both the ability to solve a larger class of problems and to test different solutions to the same problem.

3. *Computational Efficiency:* The ability to partition a problem into subproblems and to solve the subproblems often leads to considerable efficiency. It is usually efficient both in specification and solution if the solutions to the subproblems can be combined to give the over all solution. The strategy of **divide and conquer** has been used effectively in many algorithms in computer science. One of the reasons why this computational efficiency comes about is because different parts of a computational structure have different time constants. Solving the computational structure as a whole often forces the solution mechanisms to use the smallest time constants while partitioning enables us to compute each part at its own time constant.

4. *Reusability of Code and Models:* Many constraint methods in the literature use common solution techniques at different levels. These techniques may be at the lowest level in terms of numerical solution techniques or at some higher representations. For example, two different techniques may use the same numerical solver for a system of first order ordinary differential equations. At a higher level, two different techniques may use the same model of a rigid body specified by Newton's equations of rigid body motion. If these basic techniques and models are identified and implemented as a substrate, techniques that seem incompatible can be made to work together in the same framework.

5. *Experimentation with New Techniques and Extensibility:* With a basic substrate available, it becomes easier to experiment with new techniques. With a properly designed substrate and a good access mechanism, we can then begin to test our ideas much sooner than if the entire software environment had to be designed.

6. *Ability to Solve Complex Problems by Non-Experts:* Often some intelligence can be built into the modeling environment so that non-experts can use the environment with a reasonable degree of efficiency and usefulness. For example, the modeling environment may have the ability to choose the proper numerical technique to solve a system of differential equations based on the stiffness of the problem or to use an efficient technique in a matrix solution if the matrix involved is sparse.

The development of **VLSI** tools is analogous to our constraint unification effort. The VLSI technology provides the capability to fabricate a large number of electrical components in a small area on a silicon substrate. Besides the obvious savings in space taken by circuits, VLSI spurred the development of *cell libraries.* Cell libraries contain standard VLSI modules with well defined interfaces. These cells can be hooked together in various way to generate circuits, much more powerful and complicated than individual cells. VLSI technology has also permitted us to design and fabricate circuits that were not feasible with discrete transistors. For example, the high level of connectivity that is easily available in VLSI has made possible complex experiments in fields like neural nets. Integrated Chip technology has been used in making micro-antennas for high frequency applications, micro-probes to probe animal brain structures and micro-mechanical structures such as motors just tens of microns in size. The possibility of integrating analog, digital and mechanical components on the same structure holds even greater promise for new applications ([MEAD 89], [TAI and MILLER 89]).

In principle, we anticipate similar results from our research. First, bringing together various techniques that complement each other and second, various modeling "cells" that can be put together to generate powerful simulation environments.

1.5 Organization of the Thesis

The thesis is divided into four parts.

The first part, *Introduction*, has already presented the problem that we wish to explore and has provided an overview of our approach.

Chapters 2 to 6 constitute the second part, entitled *Theory and Concepts.* In this part we develop the concepts that provide a unified framework for simulations. In chapter 2, we discuss our concept of objects and constraints. In the spirit of object-oriented programming, we define generic objects. Other objects are created from

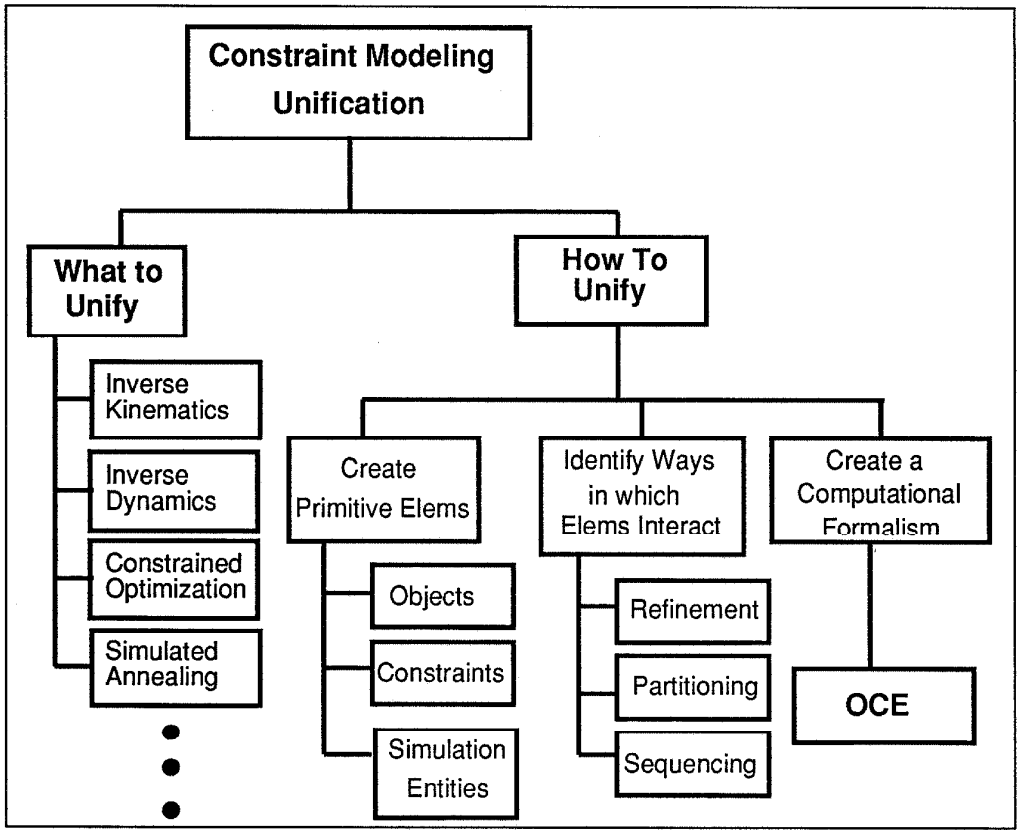

Figure 1.1: A summary of the structure of the thesis.

generic objects by inheritance. Various aspects of a constraint are also discussed in this chapter and a definition is provided. In chapter 3, we discuss different concepts of structuring in problems involving constraints. We describe three ways in which a constraint problem can be partitioned for specification and solution. In chapters 4 through chapter 6 we discuss the details of our partitioning schemes. In chapter 4, we discuss *vertical refinement.* In chapter 5, we present *horizontal partitioning* and in chapter 6, we present *temporal sequencing.*

Chapters 7 and 8 constitute the third part of the thesis, entitled *Language, Implementation and Examples.* Chapter 7 discusses the details of implementation of OCE, a prototype constraint environment built on the concepts provided in part 2 and the details of a language as a front end to OCE. In chapter 8, we present example simulations that we have carried out using OCE.

The final part of the thesis presents conclusions and appendices.

The structure of the thesis is shown in figure 1.1.

Part II

Theory and Concepts

Chapter 2

Elements of a Constraint Environment

As described in the previous chapter, we wish to bring seemingly different techniques into the same framework. This will create a unified constraint language and environment in which multiple constraint techniques can be used to solve complex problems.

In this chapter, we will define and discuss the underlying elements that form a constraint environment.

In a constraint-based modeling environment, we specify the behavior of a system of objects in terms of constraints or goals. From this point of view, objects and constraints are the basic elements in the environment which we need to clearly define. In this way, mechanisms to create and manipulate objects and constraints can be designed in the constraint language.

It is important that we create a sufficiently general definition of an object so that the constraint environment is easily extensible. We present a definition such that every entity in a constraint environment is considered as some type of object. New objects may be derived by generalizing or specializing other objects. In this way objects may be constructed hierarchically.

Constraints specify conditions on systems of objects to prescribe their desired behavior. From one perspective, a constraint may also be considered no different than an abstract object and in fact may be implemented as such. However, a constraint has enough unique features that we will present a separate definition for this type of object.

2.1 Definition of Object

The representation of an object $\mathcal{O}$ is composed of two parts,

1. A representation $\mathcal{X}$ of the state of the object. The state of an object is a set of ordered pairs

$$\mathcal{X} = \{(\text{name of property}, \text{value of property})\}.$$

   Each value in an ordered pair could be a scalar but may also be a structure such as a vector, matrix, a set, or in fact a structure of structures.

2. A set $\mathcal{F}$ of member procedures, $\{f(\mathcal{X})\}$, that

   (a) access the state of the object

   (b) modify the state of the object, or

   (c) cause the execution of member procedures of other objects

Thus an object is the ordered pair $(\mathcal{X}, \mathcal{F})$. We have chosen a representation for an object that can be structured in a number of ways from basic data types. In this way the state of an object can be tailored according to the problem being solved. As the representation of an object will vary depending on the domain of the problem, so will the manner in which the representation is used. Therefore, it seems logical to attach actions that may be performed on an object as part of its definition. In one way the data may be thought of as nouns and the associated functions may be thought of as verbs and the kind of verbs that can be applied to a noun will depend on the nature of the noun.

Why this definition?

To see the need for such a general definition of an object, we may consider a collection of different kinds of objects similar to the ones presented in [BARR 88](Figure 2.1). The simplest representation of an object may be defined in terms of its *appearance.* This representation of an object may appear as a two-dimensional image such as in an array of pixels or a set of vector line drawings.

At the next level, an object may be represented by its kinematic representation. A kinematic representation does not take into any account physical effects that may bring about that representation. The object representation is specified as a set of numbers or functions. For example, the kinematic geometric representation of a sphere is its center and radius. The time-behavior of an object may be specified by kinematic functions also. This representation of objects is indeed the one currently used, in large part, in creating computer animations.

To include physical effects, we may use a *dynamic representation* of an object. The dynamic representation takes into account physical causes of the representation, such as applied forces and torques. Consider the simulation of the motion of a rigid-body under Newtonian mechanics. The behavior of the body is not specified as a predefined

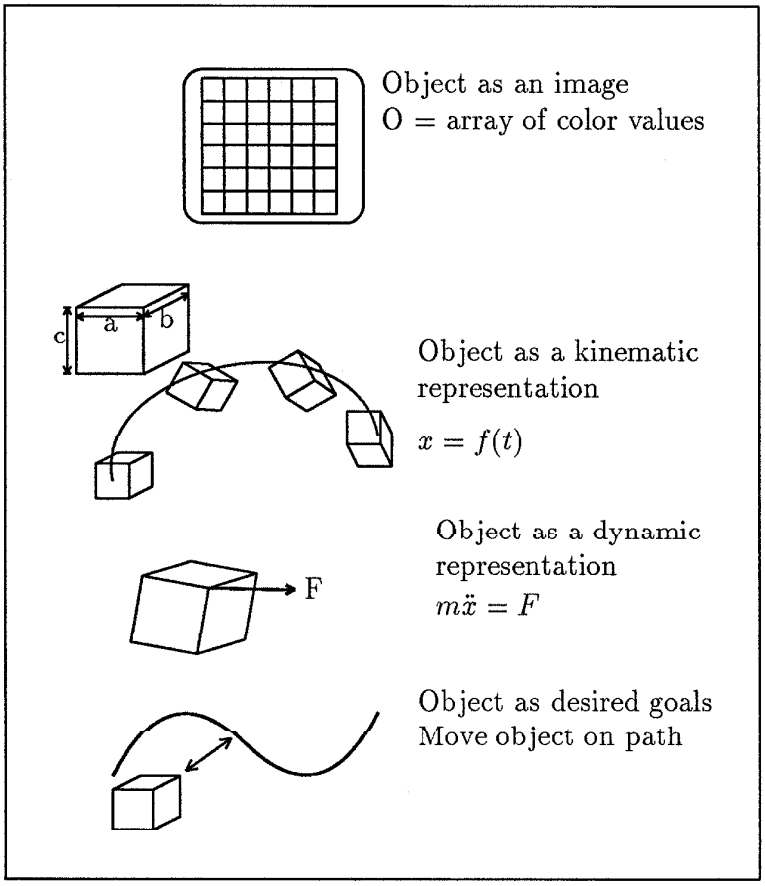

Figure 2.1: Levels of abstraction of an object.

function of time. The behavior of the body is derived from the physical causes of the motion and the physical properties of the body such as mass and moments of inertia.

The next higher level of representation of an object represents the desired behavior of the object. For example, although a mathematical representation of a polynomial may be its coefficients, we are interested in the values of the coefficients such that the polynomial passes through a number of given points. Similarly, for a rigid-body following Newtonian mechanics, we might be interested in determining a set of forces that would move the body along a prescribed path. Constraint methods convert these goals into the desired behavior. This representation may be called a constraint representation of an object.

Each of the above representations of an object is appropriate in particular domains. In fact, the same object may be represented using more than one of the above representations and representations may be changed from one to another. For

a general system of objects with many types of objects and many possible representations of an object, we need a general and encompassing definition as presented at the beginning of this section.

2.2 Definition of Constraint

Objects form a basic entity in a constraint environment. Constraints prescribe the desired behavior of the objects.

A constraint contains two main parts, a *declarative* part $\mathcal{L}$ and a *procedural* part $\mathcal{P}$.

A constraint $\mathcal{C}$ is the ordered pair $(\mathcal{L}, \mathcal{P})$
where

- $\mathcal{L}(\mathcal{X})$ is the declarative part of a constraint. $\mathcal{L}(\mathcal{X})$ specifies a logical function of the state $\mathcal{X}$ of a system of objects which evaluates to **true** to signify that the constraint has been met.
- $\mathcal{P}(\mathcal{X})$ is the procedural part of a constraint. $\mathcal{P}(\mathcal{X})$ is an algorithm or procedure that generates a state instance $\mathcal{X}_s$ such that $\mathcal{L}(\mathcal{X}_s) = \textbf{true}$.

Complex constraints with their $\mathcal{L}$ and $\mathcal{P}$ functions can be created by combining the simpler $\mathcal{L}$ and $\mathcal{P}$ functions. A unified constraint environment provides general mechanisms for the specification for $\mathcal{L}(\mathcal{X})$ and $\mathcal{P}(\mathcal{X})$.

Constraints are conditions on the state of objects comprising a system. Solving for constraints involves finding an instantiation for the system state that satisfies the set of constraints. Note that we have quite a general definition for state; the state might, for example, include a representation for the time dependent behavior of an object.

Components of a Constraint

A constraint may be considered to contain several components embodied in the above definition.

Description of a Desired Behavior or State: Constraints may specify the desired behavior of a system of objects in general terms such as a "non-mathematical" one in the form "Object A should be **on** table B" or "Object A should **follow** a path P such that object A **does not collide with** object B." On the other hand we may use mathematical relations such as

$$f(x) \quad \stackrel{c}{=} \quad 0$$

$$\begin{aligned} \mathbf{g}(\mathbf{x}) \quad &\prec \quad 0 \\ \text{minimize} \quad &h(\mathbf{x}), \quad \mathbf{x} \in \text{some domain } \mathcal{D} \end{aligned}$$

to specify constraints. The first expression specifies that $\mathbf{f}(\mathbf{x})$ is constrained to be equal to 0. The second expression specifies that $\mathbf{g}(\mathbf{x})$ is constrained to be less than zero. The third constraint stipulates that $h(\mathbf{x})$ be minimized over a domain $\mathcal{D}$ for $\mathbf{x}$ such as $\mathcal{D} = \{x : x \in \text{a set } A\}$.

Detection of Desired Behavior A mechanism needs to be defined and set up to detect if a system of objects is behaving according to a desired behavior.

In general, the detection of a desired behavior can be specified as a logical function $\mathcal{L}(\mathcal{X})$ of the state $\mathcal{X}$ of the system of objects as in the above definition. A system is satisfying the constraints imposed on it when the detection function $\mathcal{L}(\mathcal{X})$ evaluates to **true**.

The detection mechanism is directly or indirectly generated from the description of the desired behavior. For example, the constraint

$$\mathbf{f}(\mathbf{x}) \triangleq 0$$

directly generates the condition

$$\mathbf{f}(\mathbf{x}) = 0$$

to indicate whether the constraint has been met. On the other hand, the constraint

$$\text{minimize } h(\mathbf{x}), \ \mathbf{x} \in \text{some domain } \mathcal{D}$$

indirectly requires an auxiliary logical function $\mathcal{L}(\mathcal{X})$ to be derived.

Deviation from the Desired Behavior A common mechanism used in constraint satisfaction mechanisms is to design a deviation function, a measure of how far a system of objects is from its desired state. The solution mechanism attempts to reduce this deviation to zero. For example, if we wish a point $\mathbf{P}_1$ on a body B to be connected to a stationary point $\mathbf{P}_2$, the distance function between $\mathbf{P}_1$ and $\mathbf{P}_2$ measures the deviation of the constraint and the desired behavior is achieved when the distance function evaluates to zero.

There will be some types of constraints, however, where it is not possible to come up with a computable deviation function. For example, for a constraint that stipulates finding a *global* minimum of an energy function, it is difficult to find a deviation function which when reduced to zero will attain constraint satisfaction. In this case we need to use a more general means of finding when the constraint is met for the purposes of the problem and might need to use the general detection mechanism stated in terms of a logical function of state.

Procedure to Achieve Desired Behavior When a system of objects does not currently satisfy its constraints, we need a mechanism to find a state in which all the constraints are met. It might be possible to use the same constraint mechanism for all the constraints in an environment if all the object and constraints in the environment are sufficiently homogeneous. It will not be possible in general to have the same solution mechanism for all constraints; each constraint will need a solution method prescribed for it. Finding this mechanism will usually involve using a variety of different mathematical and numerical techniques. A large part of the work presented in this thesis in the next few chapters describes schemes to design such solution mechanisms.

2.3 Simulation Entities and Representations

We use the term "simulation entity" for a system of objects whose behavior we are interested in modeling. A simulation entity is constructed out of objects and constraints on the objects. For example, a system of differential equations, a collection of interconnected rigid bodies, a robot workcell, fluids or gases flowing through a pipe are all simulation entities. During the process of modeling, we might decompose a simulation entity into many simpler parts each of which would themselves be simulation entities in their own right.

The *representation* of a simulation entity is a mathematical or computational description that can be used to make a computable model.

A representation transformation mechanism transforms one representation of a simulation entity to another representation on the way to obtaining the final representation that we desire. For example, an optimization routine is a solution mechanism that transforms a representation of the state space of a problem into a member of the state space with optimal properties. A solver for first order differential equations is a solution mechanism that transforms the differential equations into functions of the independent variable.

2.4 Summary

In this chapter, we have presented our definitions of elements of a constraint environment. The definitions have intentionally been general in keeping with our desire to design a unified and extensible constraint environment. Much effort will be spent in the rest of this thesis to provide mechanisms for declaration and solution of objects and constraints. The definitions presented in this chapter provide us the guidelines for this design effort.

Chapter 3

Designing a Unified Constraint System

As discussed in chapter 1, constraints permit models to be described in terms of "desired behaviors" and in terms of goals for the models to achieve. The constraint-based approach enables a user to describe behaviors of models at a high level in a compact form: most of the computational work is off-loaded to a computer. Some techniques, such as inverse dynamics, Lagrangian physics and constrained optimization have already been introduced to computer graphics and other techniques, like simulated annealing hold promise to convert high level behavior descriptions into the parameters of a simulation entity.

In chapter 1, we described the need to consolidate constraint techniques into a unified framework. The unification of techniques will provide a means to structure the specification and solution of constraint problems and may lead to computational efficiency. The unification will lead to the development of a simulation assembly language. Using the building blocks of the simulation assembly language, complex simulations may be built. The unification will also lead to savings in implementation by sharing code and models, since a number of techniques use similar low level solution methods. With a well designed substrate for a constraint framework, it will become easier to experiment with new techniques and to incrementally develop new methods.

In this chapter, we present our design of a unified constraint system. We begin by examining a few examples of constraint problems that we wish to simulate. The examples provide an idea of the range of constraint problems that the constraint-modeling environment should be able to handle. We then present on overview of our approach. Finally, we discuss the benefits of the unified approach and how it enables us to achieve the benefits of a unified framework.

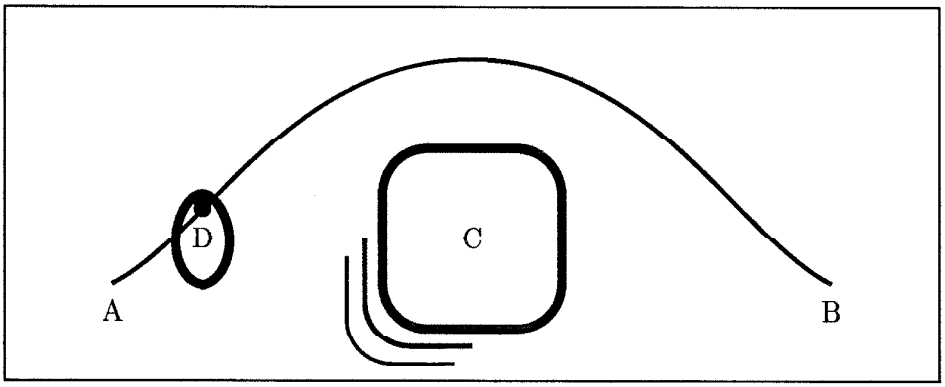

Figure 3.1: Mixing multiple solution methods to simulate a simulation entity. Object D is moved from point A to point B on a path such that it does not collide with a moving object C. The path is determined by optimization methods; the object is moved on the path by inverse dynamics.

3.1 Examples of Constraint Problems

We start with some examples of the type of constraint problems that we wish to support in our constraint-based simulation system.

Examples using Multiple Solution Methods

Our first requirement of the constraint system is the ability to combine multiple solution methods. Consider the simulation entity of figure 3.1. An object C is moving in space. We wish another object D to move from point A to point B without colliding with object C. In the example, we wish the object D to take the shortest path between points A and B without exceeding an acceleration a at any point on the path.

In the example, the motion of object C is independent of the motion of object D and the path (for instance, the motion of object C might come about from a kinematic specification, an inverse dynamics solution or an optimization.). To minimize the length of the path of object D, we could use a constrained optimization method. To keep object D on the path $\widehat{AB}$, we could choose to use inverse-dynamics techniques. Therefore, in the example, we mix multiple methods in the same overall constraint problem.

Figure 3.2 represents another example of using multiple solution methods. We wish to obtain a smooth shortest path passing through n points. The problem could arise in a robotics path planning problem in which a mobile robot has to pass through n stations. In the example, a **simulated annealing** technique might be used to generate an initial path. An optimization technique that enforces the smoothness criterion may use the path determined by simulated annealing as an initial condition.

Most numerical optimization methods select a local minimum closest to the initial starting point. By choosing a good starting guess from simulated annealing, a better result might be obtained from the latter optimizing technique. The approach of one

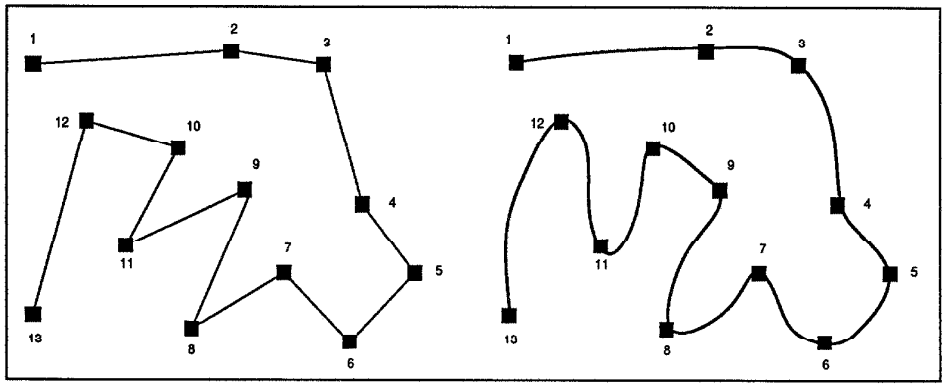

Figure 3.2: Another example of using two techniques to solve a problem. Simulated annealing is used to obtain an initial estimate of a shortest path through a set of points. A numerical technique (such as conjugate gradient) to minimize continuous functions is used to impose smoothness constraints using constrained optimization.

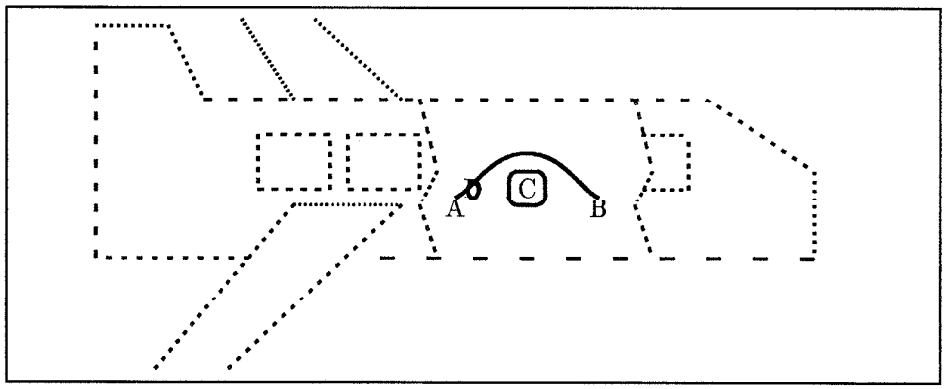

Figure 3.3: Constraints and primitives specified over multiple coordinate frames.

technique providing support and information to another technique to obtain a good solution would be a useful and general solution strategy.

Examples using Multiple Reference Frames

In physical constraint problems, different physical properties may be specified in different coordinate reference frames. The simulation entity in the example of figure 3.1 is now placed within a moving airplane (figure 3.3). The airplane is flying in a prescribed manner with respect to the ground. The constraints described in the first example in section 3.1 are now specified with respect to a coordinate frame fixed in the plane.

In the example, we see one way in which a simulation entity may be organized. Different parts of the simulation entity are specified in different coordinate frames. Therefore, we need a way to specify the frame with respect to which an object is

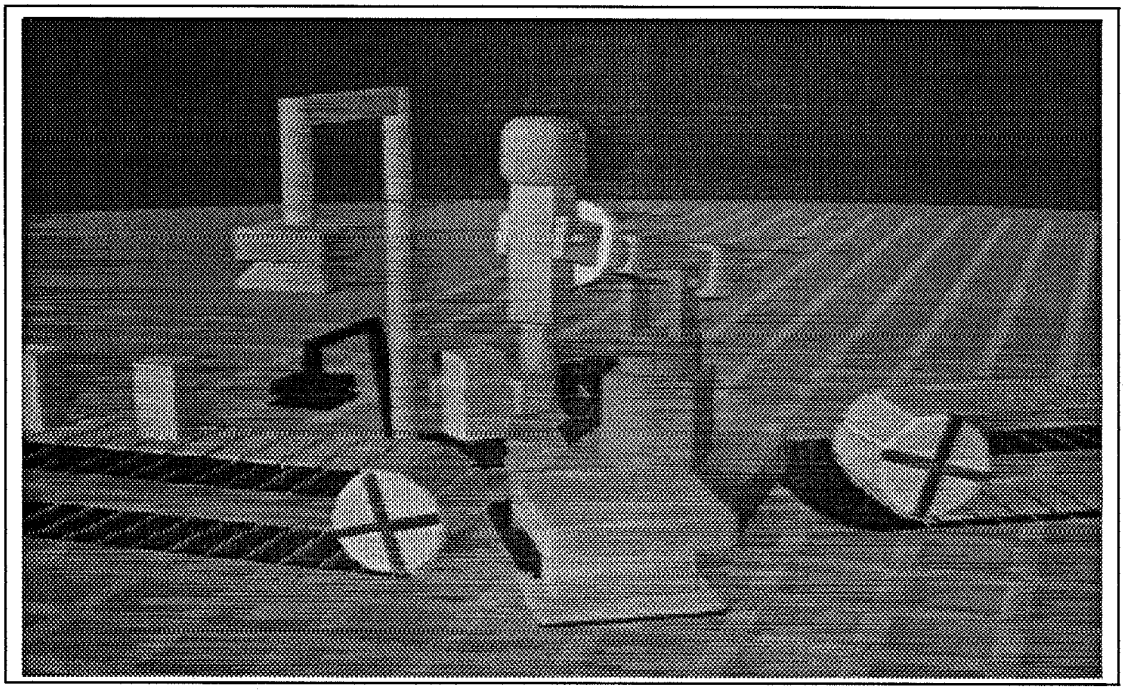

Figure 3.4: An automated workcell of a robot.

represented. We also need to relate quantities specified with respect to different frames. For instance, we might additionally impose a constraint that a point fixed in the airplane frame track a radio beacon on the ground. This leads to the need to relate the parameters of objects specified in different coordinate reference frames.

A Robot Assembly Example

Our next requirement involves interactions in the time domain. We need the ability to simulate the time behaviors of multiple simulation entities that might be interacting with each other. The behavior of a simulation entity may be affected by events occurring both inside the simulation entity and in other simulation entities.

For instance, we may wish to simulate the workcell of a robot (Figure 3.4). A conveyer belt C_1 brings work-pieces into the work cell. A computer vision system is used to indicate that the pieces are in place and to stop the conveyer belt. The robot then picks up the piece, moves it to a workbench, and inserts a pin into a hole in the piece. The robot then puts the piece on an outgoing conveyer belt C_2. The example illustrates a complex simulation entity whose time behavior can be specified in terms of time behaviors of subparts. The total time behavior of each subpart of the simulation entity may further be described in terms of multiple behavior rules. Different events occurring in the robot workcell system cause the simulation entity to

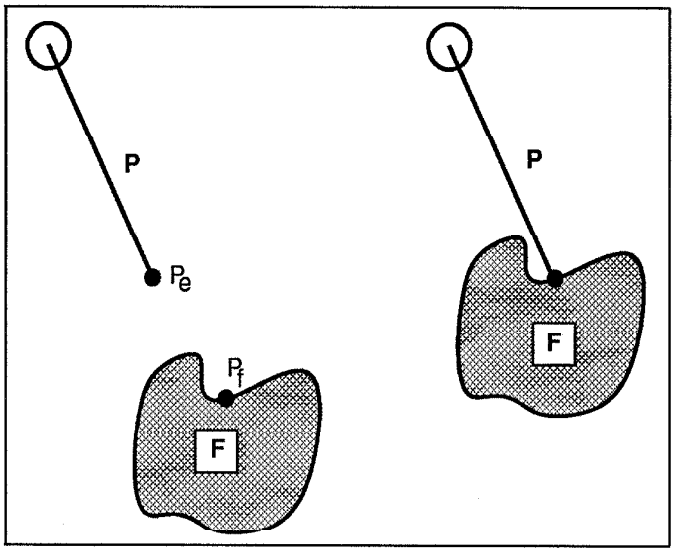

Figure 3.5: Mixing flexible and rigid objects.

switch from one behavior rule to the other.

Simulation of Heterogeneous Objects

A simulation may involve interaction of multiple heterogeneous objects. Consider the constraint system shown in figure 3.5. A flexible body F is constrained to attach to the end of an oscillating rigid bar P. In its initial position, the flexible body F is situated some distance away from the point P_e, to which body F is constrained to connect. During the simulation, the body F moves to attach to the bar at P_e. After that, the body stays attached to the bar. In the example, we need to model the elastic properties of a flexible body, rigid body dynamics and the interaction of flexible bodies with rigid bodies. In more complex examples, we may have several other type of objects such as fluids and gases, causing viscosity and turbulence effects.

Simulation of Complex Structures in Space and Time

The behavior of some systems that we might wish to simulate may be quite complex. Consider the determination of the shape and motion of biological structures such as cilia and flagella in fluid. Figure 3.6 shows a paramecium covered with cilia. The system could be simulated by computing the whipping action of a cilium and the motion of an organism under the forces produced by the interaction of cilia connected to the organism with the fluid surrounding the organism. A model of similar fluid dynamic motion was presented in [BARR 84]. To be able to model such complex systems, we need powerful specification and solution techniques in our simulation environment.

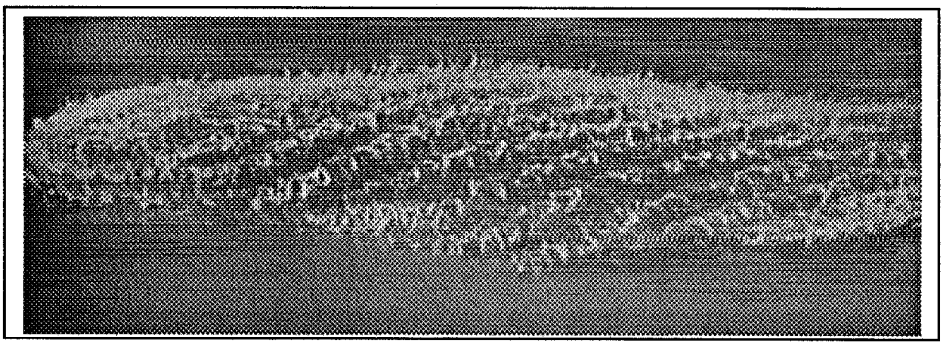

Figure 3.6: A Paramecium, example of a biological organism whose motion and shape might be simulated.

3.2 Our Approach for Unification of Constraint Techniques

The examples described in the previous section provide instances where the use of multiple constraint approaches to different parts of a simulation problem would be useful. As discussed in chapter 1, a number of constraint techniques have appeared in the literature. These techniques include inverse kinematics, inverse dynamics, constrained optimization, and Hamiltonian (and Lagrangian) physics. We observed that most of the work in constraint-based modeling involved solving a single type of problem or a narrow class of problems. The modeling environments were frequently based just on a single solution technique. Most modeling environments were implemented in a manner that was specific to the techniques or objects that the environment could handle. As a result, it was difficult to bring new techniques into their frameworks limiting the extensibility of most modeling environments.

We looked at the design of a simulation system in a general manner to provide facilities in a technique-independent way. As stated in chapter 1, we broke down our design approach into the following steps:

(Step A) Identify the set of underlying elements that constraint techniques operate on

(Step B) Identify the ways in which different elements interact with each other

(Step C) Create a computational formalism of the underlying elements that lets us use these techniques together

In the previous chapter, we discussed step A, the elements of a constraint system in the previous chapter. In particular, we defined a *simulation entity* as a system of objects whose behavior we are interested in modeling. A simulation entity is composed from objects and constraints. We now identify general ways in which

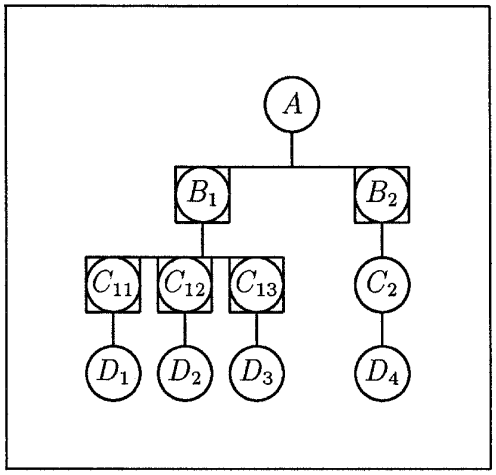

Figure 3.7: Change of representation of a Simulation Entity. An initial representation is transformed to a final desired representation using refinement and partitioning. The circles enclosed with squares arise due to partitioning. The circles without enclosing squares arise due to refinement.

different elements of a constraint system interact in the process of solution to a simulation entity.

Specification and Solution of a Simulation Entity

The modeling process of a simulation entity starts with a definition or abstraction of the system that we wish to study. The abstraction process involves choosing the aspects of the system that we consider to be important in our study. For example, to model a robot, we might deliberately neglect the flexion of the robot arms and decide to model each robot arm as a rigid body.

The result of the abstraction process is a simulation entity which we would then simulate. We need to choose a computational representation for the simulation entity that is appropriate to the abstraction we have decided to use. To find the unknown attributes of a simulation entity, we successively transform one representation into another representation. For example, suppose that we wish to model the motion of a body that we have abstracted as a rigid body, and our lowest level solution mechanisms are numerical solution routines. We may start with Newton's laws of motion as a representation of the motion of the body. This representation might be transformed into a set of first order ordinary linear differential equations. The differential equation representation would then be transformed into a representation such as a data structure that can be used by a numerical differential equation solver to generate the motion of the rigid body.

[BARZEL and BARR 90] also discusses ideas similar to the above about a simulation problem.

The transformation of representation of one simulation entity may sometimes

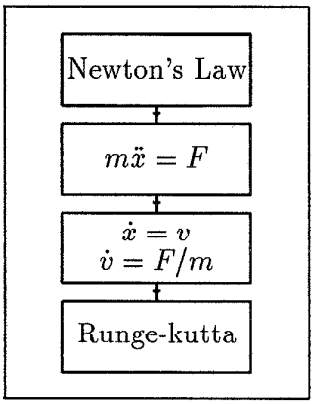

Figure 3.8: Example of constraint refinement of rigid body motion simulation.

involve creating multiple simulation subentities. Each simulation subentity would then be solved individually. During the specification and solution of a simulation entity we may conceptually create a tree-like structure (Figure 3.7).

The process of representation change is a basic solution approach which is independent of the individual constraint techniques. The representation change process involves two general solution mechanisms that can be used in constraint-based systems; these are *Refinement* and *Partitioning*. We have additionally defined a solution design mechanism that we call *Temporal Sequencing* to design time-behaviors of simulation entities.

Refinement

Refinement changes one computational representation of the simulation entity into another representation, typically closer to the primitive solution mechanisms in the underlying constraint environment. We start with a "high" level representation of the solution and refine the solution by generating representations that contain increasing amounts of detail. Typically, "lower" level of representations are also described in terms of more primitive constructs as compared to the higher level representations. In figure 3.7, a refinement step is analogous to moving down a branch of the tree when a representation change does not lead to multiple simulation entities.

For example, at one level in a constraint environment, we may have a routine that solves second order differential equations by transforming them into first-order equations. The next level may have a generic solver that solves first-order differential equations. One level further down may contain a numerical method such as a *Runge-Kutta* differential equation solver. The rigid body motion example above could then be solved by step-wise refining through these layers (Figure 3.8).

We discuss the multi-layered design of refinement in chapter 4.

Partitioning

Partitioning refers to the transformation of representation of a simulation entity that generates multiple simulation subentities. If simulation entity S_1 is transformed into S_{11}, ..., S_{1n}, we would then solve the simulation entities S_{11}, ..., S_{1n} and compose these subsolutions to generate a solution to S_1. In figure 3.7, a partitioning step is analogous to the steps that generate multiple circles on a horizontal line.

For example, consider the path planning problem of an object D moving on the shortest path from a point A to a point B as presented in figure 3.1. The problem may be broken down as

1. Obtain the path of object D from point A to point B

2. Move object D on the path obtained in step 1

Different subentities may be simulated by different techniques. Since the solutions of the subentities have to be combined, the various solution modules in an environment need to have well-defined and general interfaces so that they can be plugged together. In the example, the partitioning will work only if there is a general model of a path that can be generated and used by different techniques in the constraint environment. A simulation environment will have facilities that let a user compose sub-modules to solve a problem.

We discuss partitioning in chapter 5.

Temporal Sequencing

Time is an important concept in computer graphics. A majority of simulation approaches obtain the behaviors of simulation entities as they evolve in time. In a simulation, the behaviors that are continuous in time are as important as the events that may occur and cause discontinuities. Collisions between objects is such an example. We propose temporal sequencing as a scheme to partition the time-behavior of simulation entities during an interval of time into rules of behavior during subintervals of time. Temporal sequencing may be treated as a scheme analogous to the "push-pop" geometric hierarchies in traditional computer graphics [FOLEY and VANDAM 82] but in the time domain. A simulation entity switches from one behavior to another due to occurrence of events. Temporal sequencing provides a way to integrate events and continuous behaviors in a uniform frame work. Time behaviors may be combined using hierarchical organizations. Multiple simulation entities may be simulated with events in some simulation entities causing effects in other simulation entities. In one sense, temporal sequence may be considered as partitioning in the time domain.

We discuss temporal sequencing in chapter 6.

3.3 Considerations in the Design of a Constraint-Based Simulation System

The above approach was motivated by the goals that we want to achieve in the design of our simulation environment and the type of users that we think might find our constraint environment useful.

3.3.1 Our Goals for the Simulation System

We want to provide the following features and capabilities in our environment:

Ability to use multiple techniques: In a complex simulation, many disparate techniques will generally be used. Our simulation system should have multiple techniques available and the various techniques should have the capability to interact in a complimentary manner so that they can be brought upon to solve various parts of a complex problem.

Ability to solve problems on different scales of complexity: The simulation system should provide the capability of both setting up simple experiments of small systems and also to design big simulation systems on top of the substrate provided. For example, a user should be able to observe the behavior of a rigid body under user-specified forces. On the other hand, it should be possible to write a full-fledged robot work-space simulation system that lets a user program a robot to carry out complex tasks.

A Low Level Assembly Language for Simulations: A high level computer language is generally translated into an "assembly language" which is made up of simple primitives. The assembly language model permits the design of sophisticated high level languages by providing basic low level building blocks that can be put together in various ways. We want to provide an analog of an assembly language on top of which various simulation systems can be written. We believe that the building block approach is especially important for an extensible modeling environment supporting many heterogeneous techniques. It is usually difficult to predict what different simulation techniques will be used or added to the simulation environment and therefore it is difficult to come up with a comprehensive set of high level techniques. On the other hand, providing building blocks so that high level solution methods can be assembled by combining the building blocks in various ways provides a great degree of flexibility.

Multiple Levels of Representation: Depending on the scope of a problem at hand or the training of a user in a field, the user would like to know the least possible

about the details of a simulation environment before he can use it. The simulation environment should provide various levels of representation and detail-hiding for different levels of users. For example, a high level interface should be provided so that a user can "wire up" a quick simulation with minimal programming. On the other hand, a user should have access to the internal details of the environment if he wishes to change the environment in a big way or to add new low-level facilities to the simulation environment.

Features of User-friendliness without being Expert-hostile: Naturally, the simulation environment should be user-friendly. However, as the expertise of the user grows, the user should not feel restricted by the limitations imposed by the environment. We think that many user-friendly systems serve their purpose well for the novice user in terms of getting him started towards problem solution quickly. However, the expert user quickly becomes confined by restricted prescriptions.

Ease of Extensibility: The simulation environment is designed to be used with a multitude of techniques and objects. It is difficult to predict at the outset which techniques will ever be required by a user. New techniques may be added to the environment. For the simulation environment to be useful over a length of time, it should be easily extensible. The assembly language concept above is one aspect of providing extensibility for the environment.

Using a basic building block approach with the ability to plug the blocks in different ways provides a design that allows the above goals to be met.

3.3.2 Kinds of Users

In the design of any system, it is important to consider the categories of users who might use the system. Our objective is to design a simulation environment that accommodates a large variety of users. Some of the types of users who might find our environment useful are:

"High" Level Users: These users use specialized user-interfaces to solve specialized problems. The user interfaces may be interactive such as mouse based interfaces with various kinds of feedback including visual, tactile and force feedback. The high level user is interested in solving a specific problem without much concern about the details of the solution of the problem. The details of the simulation environment are hidden from the high level user. We do not expect a great deal of computer knowledge from this category of users.

Application Designers: These users design solution packages for various classes of problems using the facilities provided in a simulation environment. They need

to know about the various system-level facilities provided. The application designer usually interacts with a simulation environment at the level of calling subroutines and some simple programming in a provided textual language to interface to low-level facilities. Application Designers design high-level user interfaces to abstract the details of the solution process from high-level users.

System Programmers: These users extend the environment by writing low level code. They use various low level simulation language constructs and also write in various general purpose computer languages. The new code is interfaced with the existing environment through well defined interfaces and language bindings. System programmers usually program in a base general purpose computer language.

The same person might fit into a different user category at different times in the process of the solution of a problem.

As a simple example of user categories, consider the problem of designing a constraint-based curve editor. At the highest level, a user sees a menu based system where the user can use a mouse to pick curves attach constraints between objects and command the system to solve for desired figure configurations. From an application design point of view, various decisions would have been made such as how curves are represented (for example piecewise cubic curves), what solution method is necessary to solve constraints (energy minimization method), how the energy function will be formed etc. At the lowest level, a system programmer would have have provided various representations for curves, various optimization solution methods (for example conjugate gradient) that the application designer would use.

3.4 Summary

In this chapter, we presented an overview of our constraint-unification approach. We presented three approaches to design building blocks that can be plugged together to create simulations. Refinement provides a transformation between one representation of a simulation entity to another representation typically closer to the final solution representation. Partitioning refers to a transformation of representation that generates multiple simulation entities. Temporal sequencing provides pluggability between time behaviors of simulation entities over subintervals of time to create a time behavior over an interval of time.

Chapter 4

Refinement of Constraint Systems

A primary goal of the thesis is to present a design of a general constraint-based modeling environment. In the modeling environment, we wish to have the capability to support multiple techniques in the same framework. We also require the modeling environment to be extensible so that new techniques can be added as they are developed. Our approach to the design of such a system is to identify basic solution mechanisms for any constraint problem, as opposed to basing the design on solutions of particular instances of a constraint problem. One of the effective problem solution techniques is the stepwise transformation of an initial representation of the problem into a representation that can be solved by the basic solution mechanism in an environment. This transformation technique is frequently known as *refinement*. In this chapter, we shall discuss the use of refinement in forming an effective design strategy for a constraint environment.

After studying previous work and implementing a number of constraint systems, it became apparent that different systems required similar categories of methods. Most systems are just not implemented in a manner that these categories are clearly defined and have compatible interfaces. In most cases, the entire constraint process, from specification to solution, was implemented in a tightly coupled fashion. Even though the system might be using a general technique, the system was usually implemented in a very application-specific manner. The tightly coupled application-specific implementation scheme made it difficult to interface a constraint system to other constraint techniques or even to make significant modifications in techniques used in the implemented framework.

Refinement Layers for Constraint Systems

The refinement approach introduced in chapter 3 can be used to structure the design of simulations. To provide facilities in our constraint environment to use refinement as an effective solution strategy, we have identified a number of layers through which a typical constraint problem may be refined. These layers range from a "high" level

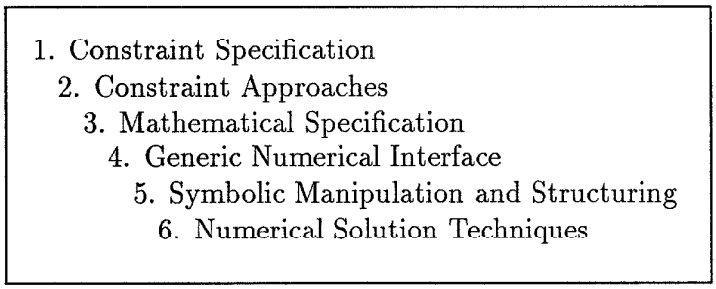

Figure 4.1: A refinement structure for a constraint-based modeling environment.

specification to basic numerical solution routines, the most primitive level in our environment. We have identified six layers of representation that we find useful in thinking about constraint-based modeling problems.

These layers are shown in figure 4.1 and 4.2.

The layered refinement structure is useful for the design of a general constraint environment from a number of viewpoints. First, the constraint refinement structure provides a guideline to design a substrate upon which various constraint methods can be built. The routines in the layers in figure 4.1 can be built with general interfaces and can be used with a variety of dependent techniques. Second, as new techniques are incorporated into the constraint environment, they can be built upon the existing routines in the substrate rather than having to implement each technique afresh. Third, as new routines are written for the substrate, their position in the above layered structure suggests the requisite interfaces that they should provide so that the new routines can be plugged to multiple other routines.

For example, most techniques using dynamics involve the solution of ordinary differential equations. Optimization techniques like gradient descent also result in a set of differential equations. A large number of implementations of such techniques have ended up using Euler's method to solve their differential equations [HAUMANN and PARENT 88][SIMS and ZELTZER 87]. Euler's method for solution of differential equations is known to have bad convergence properties. However, Euler's method is a popular method for differential equations because it easy to code[1]. A well designed substrate would make any solution method easy to use and the selection of a method could then be based on the appropriateness of the method to the problem at hand. Based on the layered refinement structure presented in this chapter, an effective computational substrate can be built that will provide a wide range of solution approaches and techniques.

A layered approach for the design of different representation levels of simulation entities has another advantage. Support will usually exist in the lower layers to be able to build a custom solution conveniently, even if a high level solution technique

[1]We strongly recommend that users NOT use Euler's method!

is not already implemented.

Hence, although refinement is a popular existing problem solving strategy in computer science, the application of the refinement technique to the design of a general constraint environment as the above delineation of layers is a new contribution to computer graphics modeling.

4.1 A Layered Structure for Constraint Systems

In this section we will discuss each of the six refinement layers in figures 4.1 and 4.2. We present examples of some basic techniques that can be implemented in each layer and may form a useful computational substrate. The choice of examples of basic techniques has been based mostly on the study of existing constraint systems (and our experiments), but the layered design does not depend on any particular implementation.

4.1.1 Layer 1: Constraint Specification

The solution of a simulation problem starts with a high level specification of desired behavior goals, for example,

- Move object A on path P in a physical environment such that object A spends the least amount of energy
- Find shape of body B under forces F1 and F2 given elastic properties of body B
- Find the motion of a flagellum in fluid of given viscosity
- Plan the path of a robot arm to move object A avoiding obstacles in a scene

At this first level of specification, we are not concerned about the final solution methods that will be used or required to obtain the desired behavior. The initial specification will need to be refined into lower layers to find a final solution mechanism. The ability to use high level specifications of constraint problems makes it easy for users to avoid having an intimate knowledge of the implementation details in the simulation environment. For example, the statement:

```
compute_shape(elastic_body B, Force f1, Force f2)
```

might be sufficient to generate a complete description of the shape of body B in the second bulleted example above. The fact that the `compute_shape` process might involve the solution of partial differential equations is not important at this level of specification (and might not be of interest to the user).

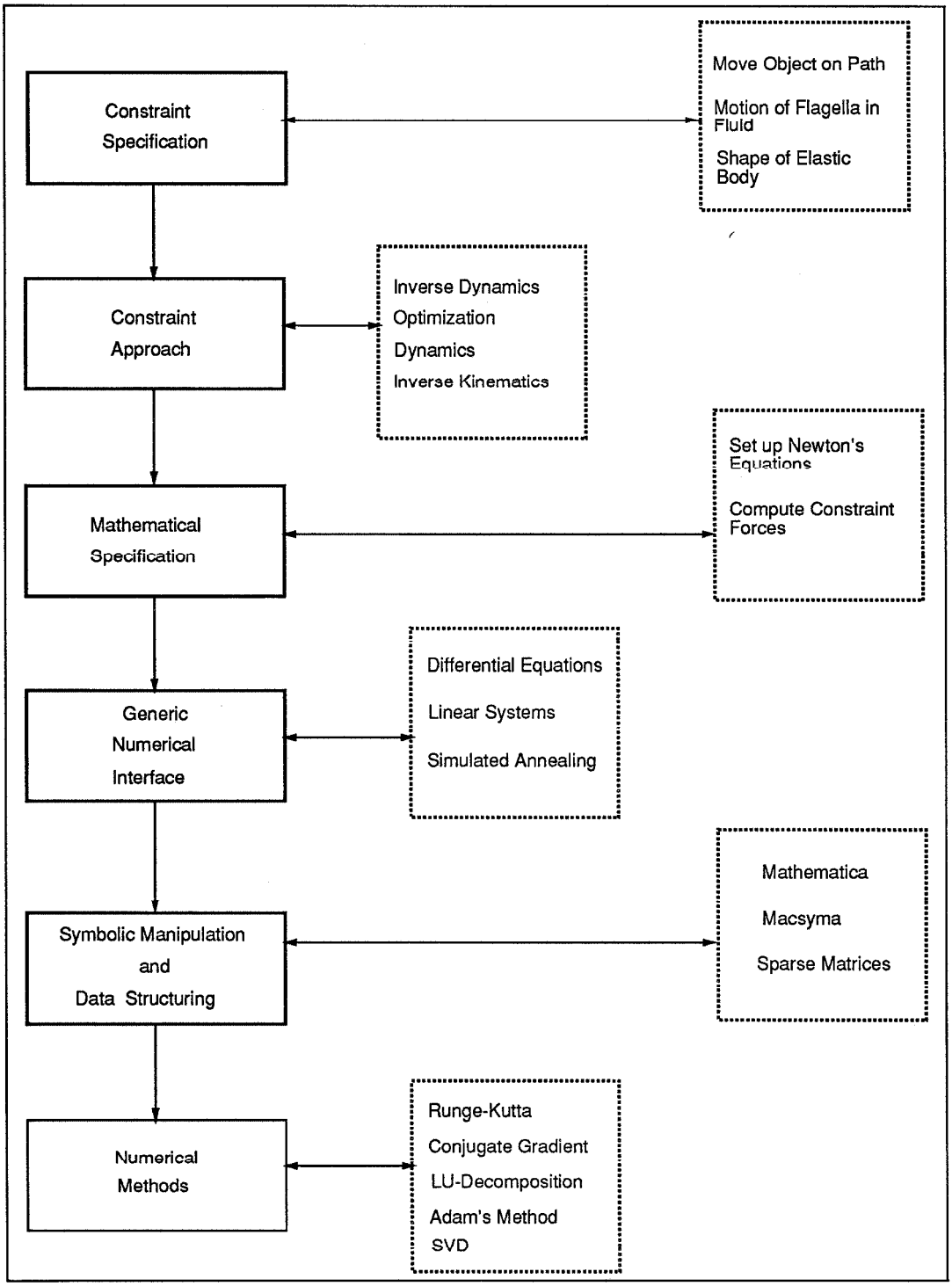

Figure 4.2: Layers of refinement in a simulation system.

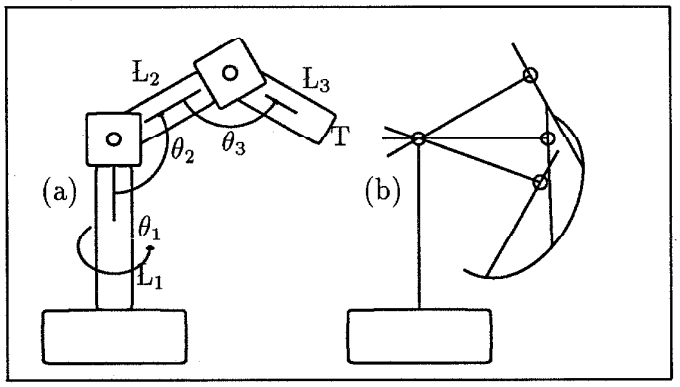

Figure 4.3: A three-jointed robot. The angles θ_1, θ_2 and θ_3 control the position of the tool T. Part(b) shows three configurations of the robot computed by inverse kinematics such that the tool T follows the specified trajectory.

4.1.2 Layer 2: Constraint Approaches

The second constraint-refinement layer provides a collection of different constraint approaches to solve the constraint problems (or parts of the constraint problems) posed in the first layer, constraint specification. The various constraint approaches mentioned in section 1.2.1 may be used in combination to compose an overall solution strategy.

Several of the methods which are found useful in a constraint-based environment are described below. These methods are just a sample from currently used techniques in constraint-based modeling. These techniques are presented here to aid in the choice of numerical techniques to be implemented in lower layers.

Inverse Kinematics

Kinematics refers to the study of the motion of bodies with no consideration of the physical causes of the motion, i.e., physicsless motion [SHAMES 82].

For example, the complete motion of a rigid body may be specified geometrically by specifying

$$\begin{aligned} \mathbf{f}(t) &= \text{position of origin of body} \\ \mathbf{R}(t) &= \text{rotation matrix with respect to world coordinates} \end{aligned}$$

where $\mathbf{f}(t)$ and $\mathbf{R}(t)$ are vector and matrix functions of time respectively. Thus, for a set of rigid bodies with constraints, an inverse kinematics solution is the pair of functions $\mathbf{f}(t)$ and $\mathbf{R}(t)$ for each of the bodies such that the specified constraints are met.

As another instance, consider a multi-body linkage such as a robot in figure 4.3a. For this robot, the values of the angles θ_1, θ_2 and θ_3 determine both the configurations

of the three arms of the robot and the position of the tool T on the end arm. The kinematics of the point T can be specified in terms the three joint angles:

$$\mathbf{P}_T(\mathcal{X}) = \mathbf{f}(\mathcal{X}_1, \theta_1, \theta_2, \theta_3)$$

where, $\mathcal{X}_1$ represents the fixed parameters of the robot such as the lengths of the arms and the position of the base.

Suppose that we wish to impose constraints on the path of the tool carried by the robot. In this instance, given that the end point T of arm L_3 has to follow a given trajectory, the inverse kinematics problem would be to determine the variation of joint angles θ_1, θ_2 and θ_3 as functions of time that would cause the desired motion of the tool T (Figure 4.3b).

Forward Dynamics

Forward dynamics involves the determination of motion of bodies under the influence of applied forces and torques for rigid, flexible, or fluid objects. The motion of rigid bodies is determined by using Newton's equations of rigid body motion that relate linear and angular accelerations of bodies to the forces and torques applied through inertial properties of the bodies, namely, mass and the inertia tensor [GOLDSTEIN 80]. The equations of rigid body motion are second order ordinary differential equations. Similarly, equations from elasticity theory can be used to determine motion of flexible bodies and Navier-Stokes equation may be used to model fluids.

Forward dynamics may be extended to compute "motion" of generalized variables. In this case, a system configuration is described in terms of generalized variables, one variable corresponding to one degree of freedom of the system, and generalized forces that cause the values of generalized variables to change. For example, for the system in figure 4.4, the generalized variables are angles θ_1 and θ_2. The forward dynamics problem is to determine the variation of the angles θ_1 and θ_2 starting from prescribed initial values under the force of gravity. One of the methods to determine the generalized equations of motion of systems is by **Lagrange's method** [GOLDSTEIN 80] that generates differential equations relating the derivatives of expressions representing energy of the system.

Inverse Dynamics

Inverse Dynamics is the inverse problem of Forward Dynamics. Inverse dynamic techniques compute the necessary forces and torques on a body that will result in a desired motion or equilibrium state. The forces and torques are computed by using the specification of the motion and the forward dynamics laws that apply to the bodies.

As an example of constraints using inverse dynamics, consider the system of bodies in figure 4.5. The system consists of two bodies B_1 and B_2. A point P_{11} on body

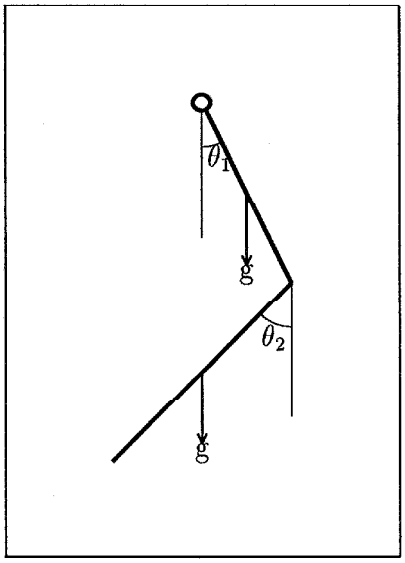

Figure 4.4: Forward dynamics in terms of generalized variables. Lagrange's method can be used to determine the equations of motion of the complex pendulum under the force of gravity in terms of the generalized variables θ_1 and θ_2.

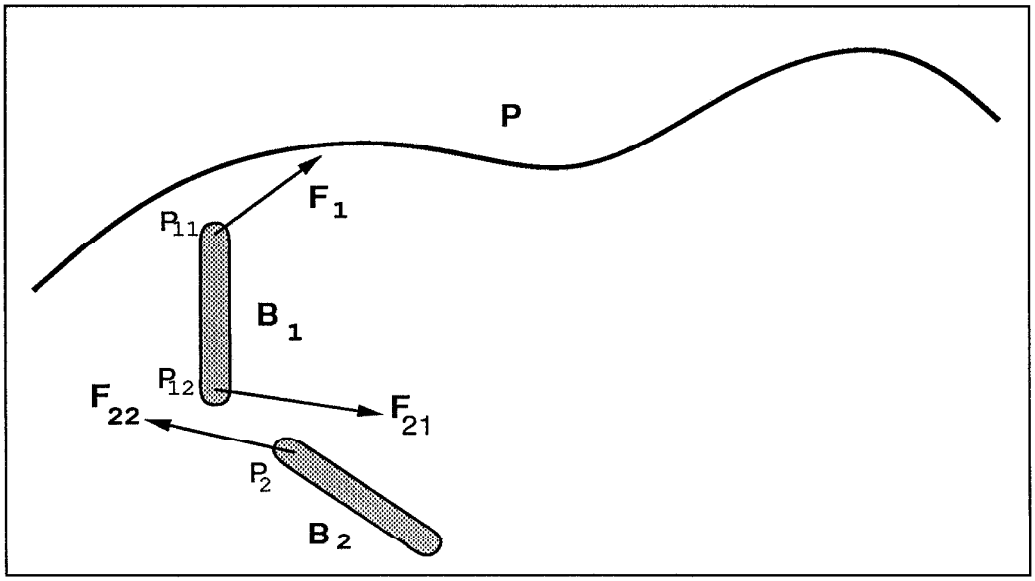

Figure 4.5: An example of Inverse Dynamics. We need to compute forces F_1 and $F_{21} = -F_{22}$ that would move bodies B_1 and B_2 under specified constraints of path following and interconnection.

B_1 is constrained to be on a prescribed path $\mathcal{P}$. At the same time body point P_2 on body B_2 is constrained to be connected to point P_{12} on body B_1. The bodies are assumed to move according to Newton-euler equations of rigid body motion. The inverse dynamics problem involves finding the constraint forces F_1 and $F_{21} = -F_{22}$ as a function of time that will move the bodies according to the above constraints.

Optimization techniques

Optimization is a process of minimizing or maximizing an *objective function*, possibly in the presence of some constraints.

Many computer graphics constraint problems can be cast as a constrained optimization problem involving an energy function of the state of the system [WITKIN, FLEISCHER & BARR 87] [PLATT 89] [KALRA 90b]. In principle, a state that minimizes this energy function is the state that satisfies the constraints imposed on the system.

4.1.3 Layer 3: Mathematical Specification

A constraint approach such as described above, specifies a general overall mechanism towards the solution of a constraint problem. Once a constraint approach has been selected, we need to generate a representation of the constraint problem that can be solved mathematically. The mathematical specification is a formulation of the problem in terms of mathematical structures such as sets of differential equations which when solved will generate the solution to the constraint problem.

For example, consider the following approach to inverse dynamics. In the context of rigid bodies, the approach solves for forces which (when applied to the bodies) will produce a desired motion of bodies. To be able to compute the solutions with this technique, a mathematical representation may be generated as in [BARZEL and BARR 88], via:

1. Form a "deviation function" for each constraint

2. Compute constraint forces for the objects in the system such that each deviation function is reduced to zero using a critically damped trajectory.

3. Integrate the differential equations of motion of the objects using the constraint forces and external forces in the system

A complete elucidation of the above example is provided in [BARZEL and BARR 88] where the first two steps result in a linear system of equations for the computation of forces.

Similarly, to use optimization as a constraint-solution technique, we need to form an "objective function" and the constraints functions of the problem. More precisely,

an optimization problem may be written as [GILL et al 81]:

$$\begin{array}{lll} \text{minimize} & F(\mathbf{x}) & x \in \Re^n \\ \text{subject to} & E_i(\mathbf{x}) = 0, & i = 1, 2, ..., m \\ & I_i(\mathbf{x}) \geq 0, & i = 1, 2, ..., p \end{array}$$

In the above formulation, m or p or both may be zero. If $m = p = 0$, the problem reduces to an unconstrained optimization problem.

The mathematical specification step generates a general mathematical representation of the problem.

4.1.4 Layer 4: Generic Numerical Interface

After the mathematical statement of the problem has been made, the resultant mathematical problems need to be solved. The generic numerical interface in a constraint-environment implements a front-end to numeric solution of generic mathematical problems. Typical front-ends might be a linear equation solver, a differential equation solver, and an optimization package.

For example, to solve a system of differential equations:

$$\begin{aligned} y_1' &= f_1(y_1, y_2, \ldots, y_n, x) \\ y_2' &= f_2(y_1, y_2, \ldots, y_n, x) \\ &\vdots \\ y_n' &= f_n(y_1, y_2, \ldots, y_n, x) \end{aligned}$$

a variety of methods may be used, such as, Adam's method and Runge-Kutta methods of various orders (even Euler's method), depending on the nature of the differential equations. Although particular numerical methods may differ in detail, each of them computes the solution to a set of differential equations, a set of n functions, $y_1(x)$ to $y_n(x)$. The purpose of providing generic numerical interfaces is to hide the details of specific numerical techniques that may be employed and to present a uniform front-end that is appropriate to the mathematical representation of a problem.

Selection of a numerical solution technique

Given a generic interface, how do we select a particular technique to solve the problem?

Each generic interface should have a default technique that would be used in the absence of any auxiliary information. Interface facilities should be provided for a user to specify specific numerical routines to effect computation in a desired manner. Examples of such implementations for generic numerical solvers are presented in chapter 7.

In some cases, it is possible for the generic interface layer to use information about the problem to choose the proper numerical routine depending on the nature of the problem. For example, if a set of differential equations is **stiff**, the layer should choose an appropriate method. Similarly, for a sparse matrix or a tridiagonal matrix, a method that takes advantage of the sparsity should be used.

An advantage of providing this layer is that users who are not experts in numerical methods can still employ the default numerical solution methods capable of solving their problems. In fact, if the generic interface layer has the capability to select appropriate numerical methods depending on the nature of the problem, non-experts can benefit from this "numerical expert-system." Experts can still override defaults and tailor the solution according to their needs. This approach is according to our philosophy: to provide a user-friendly system without being expert-hostile.

4.1.5 Layer 5: Symbolic Manipulation and Structuring

Once we have formulated our problem in general mathematical terms and have selected a method to solve the problem, some work is still required before a numerical routine can be used. This work is mostly in generating extra information from the problem statement and in organizing data into a form that is usable by a numerical routine. The symbolic manipulation and structuring layer provides this support.

Symbolic Manipulation

In most problems that use numerical methods, a number of mathematically extraneous structures must be computed in addition to the structures required for the central problem.

For example, consider the problem of minimizing a function

$$f = f(x_1, ..., x_n) \tag{4.1}$$

This unconstrained minimization problem is a clearly stated mathematical problem. To solve this problem we need to select a numerical method. Various methods may require different types of information from the above canonical statement of the problem and functions to provide this information have to be written.

One simple technique to find a local minimum of a function is **Conjugate Gradient**. This technique requires the value of the gradient of the objective function

$$\nabla f = \left[\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n}\right].$$

Some more sophisticated techniques may require the *jacobian matrix* and the **Hessian** matrix of functions also. These auxiliary mathematical structures are not

directly related to do the central problem but are computed depending on the particular technique used as a solution method. Further, most of these structures can usually be computed by a rote method such as symbolic or numerical differentiation.

We believe that this computation should be done by the simulation environment. Many good symbolic manipulation packages like [MATHEMATICA 88] and [MACSYMA 77] now exist that can be used to carry out most of the work of deriving mathematical structures required for the solution of a problem. The simulation environment needs to provide a convenient interface to such packages.

Data Structuring

During the solution process of a constraint problem using numerical techniques, effort is also spent structuring the data into the form that can be used by a numerical method. An example is setting up correspondence between indices of vectors generated by one part of the solution process with vectors used by another part. Another example is handling of sparse matrices where mechanisms need to be set up for access and modification for the sparse matrix representation. This is a very error prone process during programming and can usually be handled automatically by a computer.

4.1.6 Layer 6: Numerical Solution Techniques

From the point of view of our constraint environment, the numerical solution techniques layer is the most primitive layer in the system. This layer is composed of raw numerical solution routines that provide the solution of basic mathematical problems such as solution of differential and algebraic equations and optimization problems. These routines may be taken from numerical libraries or may be simple interfaces to these routines. Many good sources of such routines are widely available such as [NAG 89] and [PRESS et al 88].

4.2 Advantages of a Layered Approach

The above layered approach for refinement in constraint systems provides the following benefits for the design of a constraint-system.

Provides a Structure to the Design of a Large Software System

The simulation environment that we are designing and implementing is a large software system. By partitioning the environment into sub-parts that build on each other, we obtain a design structure for the implementation of the system. This partitioning also enables us to think and produce general interfaces for solution techniques so that any technique can be used by more than one dependent technique.

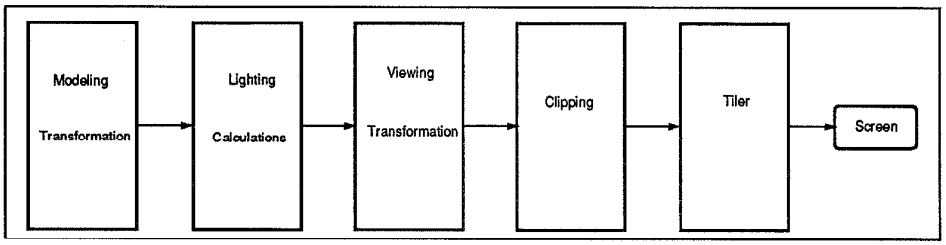

Figure 4.6: The Graphics Pipeline.

Different Entry Points into the Solution Process

As discussed at the beginning of this chapter, the refinement pipeline presented in this chapter provides various levels of abstractions to the same problem. A user may start the solution of his problem at a number of different levels depending on the how specific his problem is. Even if the exact solution to his problem does not exist, a user can still use the basic facilities that already exist in the system and build up from there.

Hardware Amenability of the layered Approach

The layered approach presented in this chapter suggests a pipeline for the solution of constraint methods. We see the possibility of efficient and fast hardware implementation of simulation systems as an important advantage of the above layered approach, although some parts of this layered structure are more amenable to hardware implementation than others. By separating various conceptual parts of a simulation system, it becomes easier to decide what parts would benefit by a hardware implementation to provide big speed benefits.

We may compare the layered approach with the **graphics pipeline**. In the 1970s, algorithms were developed to render polygons on **bit-mapped displays**. In the early 1980s, a sequence of operations was defined in order to render polygons (figure 4.6). This sequence operated on polygons in *modeling space* and through various transformations generated a two dimensional projection on a discrete grid of pixels. This pipelined delineation of the polygon rendering process spurred the development of specialized hardware that speeded up the time consuming parts of the rendering process. These developments resulted in work stations with fast rendering systems that can render hundreds of thousands of polygons per second.

There are certainly differences between the polygon rendering process and the general simulation problem. The simulation problem has many disparate techniques and objects and at different times different operations may be performed on any object. There are still some time consuming parts in our "simulation pipeline" that can benefit greatly by migration into hardware. Already there is hardware available to carry out vector and matrix operations. An attractive numerical problem to migrate

into hardware is the solution of differential equations. There is great potential in combining analog hardware [MEAD 89] and digital hardware to come up with fast differential equation solvers. Some applications of analog hardware in optimization problems were also presented in [PLATT 89]. Just like the vector and matrix operations boards available today, we anticipate the availability of plug-in boards that implement other numerical methods.

4.3 Summary

In this chapter, we presented a layered structure of refinement steps which might be employed to generate solution to a high level problem in terms of primitive solution structures such as basic numerical solution methods. Higher layers in this structure permit the use of representations that hide implementation details of lower layers. The layered approach also suggests a hardware implementation that may speed up some parts of the solution process.

Chapter 5

Partitioning of Constraint Systems

In the previous chapter, we applied refinement to the design of a "layered" structure for a constraint environment. Refinement is the step-wise transformation of one representation of the simulation entity into a representation that can use the basic solution mechanisms provided in the environment.

During the refinement process, it is sometimes necessary to *partition* a simulation entity into smaller simulation entities before further refinement can proceed. The solutions to the sub-entities are devised separately and are then composed to generate an overall solution. The partitioning process may take place at any of the vertical refinement layers presented in the previous chapter (Figure 5.1).

Partitioning is the decomposition of a simulation entity into sub-entities, and is a useful and general way to organize solution of problems. Like refinement, partitioning is a general solution strategy independent of any particular constraint technique. Partitioning may also generate a division of a problem into subproblems that may be solved concurrently. Partitioning may also make apparent a parallel or pipeline structure in the problem.

In this chapter, we identify several types of partitioning for constraint systems so that we can provide them in our constraint environment. We discuss a number of ways in which this decomposition may occur and how the solutions to the sub-problems may be combined to generate an overall solution to the problem. We also present some computational constructs that can help to effectively use partitioning in a constraint environment.

5.1 Use of Partitioning in Constraint Systems

We now present a constraint problem classification based on interdependence of sub-problems during solution. Depending on their interdependence, different solution

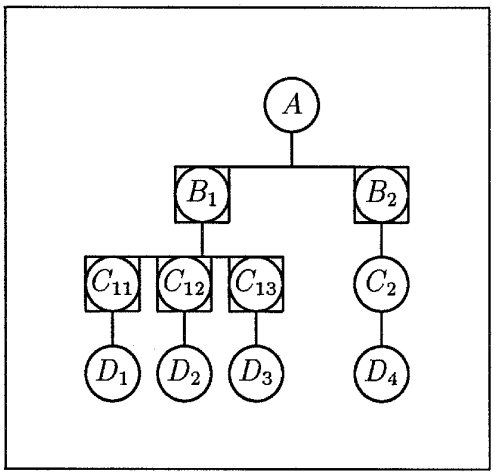

Figure 5.1: Solution of a simulation problem through a change of representation. An initial representation is transformed to a final desired representation using refinement and partitioning. The circles enclosed with squares arise due to partitioning. The circles without enclosing squares arise due to refinement.

methodologies for the subproblems are required.

We consider the solution of a partitioned problem to consist of two steps, *subproblem solution* and *subsolution composition*. The subproblem solution involves solving each of the subproblems of the main problem. The subsolution composition step combines the solutions to the subproblems, often through simple transformations, to form the over all solution.

5.1.1 Independent Subsystems

Independent subsystems do not interact during the subproblem solution process. Each subsystem can be solved by a separate solution technique and may be solved in arbitrary order. Each subproblem, however, has to be solved before the overall solution can be composed. Once the subproblems have been solved, a simple compositions converts the subsolutions into the solution to the main problem.

An example of an independent system is given in figure 5.2. This example shows the composition of a figure in a constraint based editor. We wish to form a figure with rectangles aligned on a line. A text string is centered in each of the rectangles. In the example in figure 5.2, the constraint to center the text in each rectangle can be solved completely in the coordinate frame of the rectangle. The alignment of the rectangles on the line is solved completely in the line frame. Once these constraints have been solved, a simple coordinate transformation generates the representation of each object in a common coordinate system (the screen) where each object may be rendered.

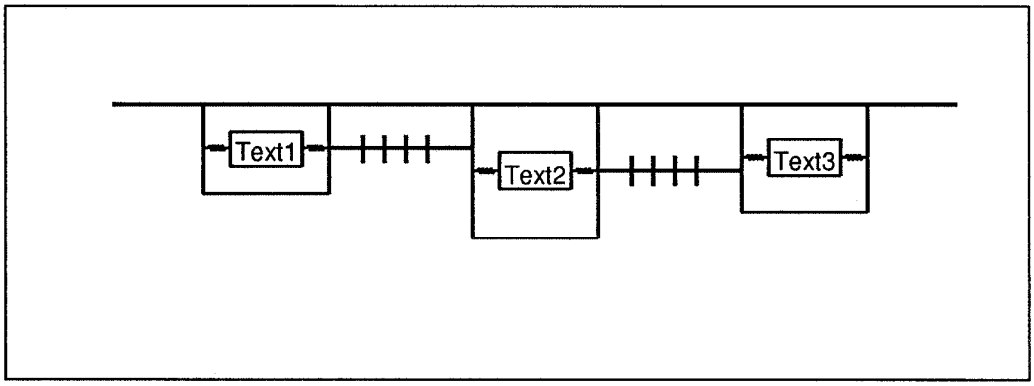

Figure 5.2: An example of independent subsystems. We wish to create a figure by centering text strings inside rectangles. The rectangles are aligned on a line. The "springs" in the figure are used as a symbol for constraints. Various subsystems in this example can be solved in arbitrary sequence and the solutions combined after all the subsystems of constraints have been solved.

Sequence independent systems are natural candidates for solution on multiple independent machines. The solution proceeds to completion with no communication. Only when the sub-solutions are complete, a master process combines the sub-solutions to generate the complete solution.

5.1.2 Sequenced Subsystems

Sequenced subsystems represent a partitioning of a problem in which one subsystem depends sequentially on another subsystem for its solution. That is, a time ordering can be created for the set of sequenced subsystems such that one subsystem may be solved completely before the solution of another part needs to be attempted. Further, a sub-system cannot be solved until its predecessor in this solution ordering has been solved.

As an example of sequenced subsystems, consider the following example. Move an object A, from point B to point C around obstacles D and E such that an upper limit of the curvature of the path is κ (Figure 5.3). This system can be broken down into

1. Compute the path from point B to point C under the specified constraints

2. Move object A on the path obtained in step 1

Note that in this example we might use two different solution techniques, e.g., constrained optimization to determine the path and inverse dynamics to move the object on that path. We have to, however, completely solve the constrained optimization part of the solution and only then can we start to solve the inverse dynamics

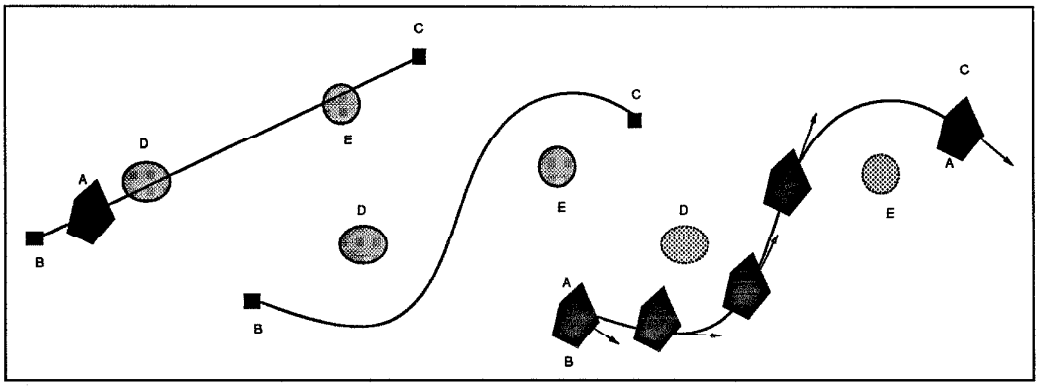

Figure 5.3: An example of sequenced subsystems. To move object A from point B to C around obstacles D and E, we can first solve for the path, and then move the object on the path.

part. Further, the inverse dynamics solution requires the solution from the optimization constrained part (the path) and therefore can not be attempted until the optimization is complete.

5.1.3 Unpartitioned Systems

If a system is not decomposed into subsystems during a vertical refinement step, we just transform the representation of the system to another representation. In some problems, it might not be apparent how to obtain a partitioning into subproblems at the current level of representation such that the subproblems can be solved separately. By refining the representation, a partitioning might appear.

Change of Representation to Form Sequenced Subsystems

As an example, consider the constraint system shown in figure 5.4. A flexible body F is constrained to attach to the end of an oscillating rigid bar P. In its initial position, the flexible body is situated away from the bar. The body F then moves to attach to the bar and after that oscillates with it. We decide to use inverse dynamics to move the elastic body to the bar and to keep it attached. To model the shape of the flexible body, we decide to use a **finite-difference method** to approximate the elastic properties of the body [TERZOUPOULOS et al 87]. In this example the behaviors of the rigid body P and the flexible body F are interdependent and cannot be solved independently of each other. The constraint forces to move the bodies couple the behaviors of the rigid bar P and the flexible body F. If we look at the problem from a lower level of representation, we can find a sequenced set of solution methods.

The steps are:

1. Determine the force between the end point P_e of the bar P and the point P_f on the flexible body F such that P_f moves towards P_e. This computation has to

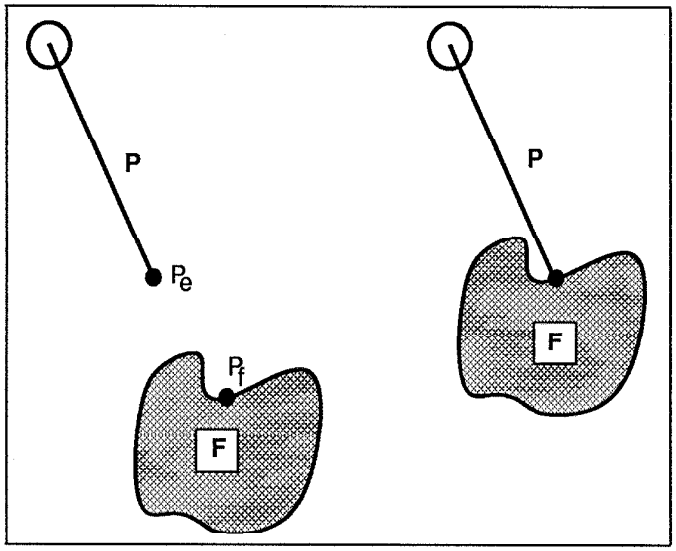

Figure 5.4: An example of system that may be partitioned into sequenced system by changing the level of representation.

take into account any other forces on body P and body F that may be present.

2. Find the motion of the rigid body P under all the forces on it

3. Find the motion of the flexible body F under all the forces on it

Note that the solution is still described at a relatively high level. Each of the above solution steps may be solved by using one of many lower level methods. By going down a level of representation we have deduced a scheme in which force computation, rigid body dynamics and finite-differences steps can be interleaved. This example will be solved in chapter 8.

5.2 Constructs for Horizontal Partitioning

Given the categories of horizontal partitioning presented above, what facilities in a simulation system could be provided to use partitioning easily and effectively? We present a discussion of some such constructs in this section.

5.2.1 Concurrent Solution

For independent systems, constructs for the parallel or concurrent solution of subproblems are required. These constructs should be implemented to take advantage of the underlying architecture of the computing environment. In a computer system that supports multiple processors, the subproblems could be solved as subprocesses

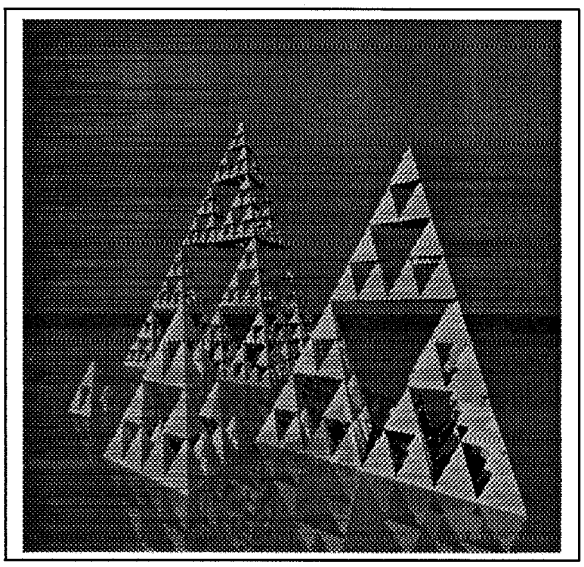

Figure 5.5: An example of hierarchies in object modeling.

spawned on different processors or machines. On a sequential machine the subproblems may be solved one after another or interleaved in a time-sharing fashion. In any case, the specification of partitioning should be independent of the details of the implementation. The specification would provide a hint to the system which could be exploited, if possible, to yield efficiency. Parallelism can be exploited at any level of representation where it exists.

5.2.2 Solution in Local Coordinate Frames

The solution of subproblems in local coordinate frames is a useful tool in the hierarchical solution of problems. Facilities to associate coordinate frames with objects and use the frames should be part of the simulation environment. The following features to deal with coordinate frames may be provided.

Declaration of frames and basic transformations: There should be a capability to associate a frame with any object in the system. Operations to transform basic quantities should be provided to transform between frames. Some useful frame dependent quantities are position, velocity and acceleration vectors and tensors of various orders.

Hierarchies of Coordinate Frames: Hierarchies arise naturally when using coordinate frames. An almost literal example of hierarchies is shown in figure 5.5. Hierarchies have been long used in computer graphics for static scene modeling

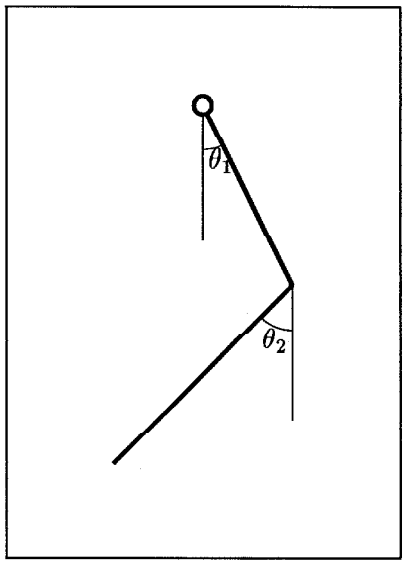

Figure 5.6: The motion of a complex pendulum is more naturally specified in terms of θ_1 and θ_2 as compared to specification in a cartesian frame. A generalized frame is useful for systems that use generalized coordinates.

[FOLEY and VANDAM 82]. The objects in figure 5.5 were modeled using such a hierarchical transformation tree.

Transformation hierarchies can represent organization of a system of objects very effectively. We are also interested in time-dependent behaviors of systems and the hierarchical structure of a system may change during simulation. As an example, consider a Rubik's cube. Each of the faces of a Rubik's cube can rotate about an axis perpendicular to it. All the sub-cubes on a face that are shared by two or three faces can move with the rotation of any of these two or three faces. Such sub-cubes that may move as part of more than face need to belong in different parts of an object hierarchy depending on which face is rotated. We need to be able to define hierarchies that change with time to simulate such behaviors.

A simpler example is a robot picking up an object A from a table T. When the robot picks up object A, object A and any hierarchy subtree rooted at object A becomes a part of the hierarchy of the robot. Object A was part of the table hierarchy before the robot picked up object A.

Generalized Coordinate Frames: Traditionally, rectangular cartesian frames are used in simulation systems. However some problems are more naturally representable in generalized frames. For example, the complex pendulum in figure 5.6 is more naturally represented in terms of two rotation angles θ_1 and θ_2. In a

general system, a user should be able to define and use generalized frames and to transform quantities between generalized frames and cartesian frames.

5.3 Advantages of Horizontal Partitioning

Why do we care to partition the problem? There are some very distinct advantages that accrue.

Use of Hierarchy: Hierarchical decomposition of the problem and hierarchical composition of the solution of subproblems may lead to a very effective solution strategy. The classical computer science strategy of *divide and conquer* is an example of exploiting hierarchy in problem solution.

Use of the Structure of Problem Solution (Pipelined and Parallel): Partitioning of a problem often suggests an effective structure to the solution of the problem. In some cases, partitioning may lead to subproblems that have very little or no coupling during some parts of the solution process. These subproblems can be solved in parallel which could speed up the solution process as a whole. If the dependence of subproblems on each other is ordered, a pipeline structure for the solution may be effective and may be exploited by overlapping solution of subproblems.

Computational Savings: In many instances, different subproblems of a problem have widely varying time constants. Solving the problem as a whole forces the solution process to proceed at the rate of the slowest time constant. If the problem can be decomposed into parts, each subproblem can be solved at the time constant of the subproblem resulting in substantial computational savings.

5.4 Summary

In this chapter, we presented how partitioning may be used as a problem solving strategy in constraint systems. We presented a classification of partitioning in constraint systems based on how the subsolutions are to be combined. The classification suggests methods and structures that can be implemented in a modeling framework to make partitioning a useful problem solving tool for a unified constraint environment.

Chapter 6

Temporal Sequencing

In this chapter, we present a general, systematic and consistent treatment of time and event modeling. We formalize the concept of events and create an object called an *event unit.* Using an event unit as a "time primitive," we present a succession of organization schemes to hierarchically create more complex time sequences or systems of events. Temporal sequencing may be treated as a scheme analogous to the "push-pop" geometric hierarchies in traditional computer graphics [FOLEY and VANDAM 82] but in the time domain. The concept of an event unit provides a partitioning scheme for the creation of a time dependent phenomenon over a time interval by letting us create and compose behaviors over subintervals.

6.1 Use of Time in Simulations

In a simulation, we are typically studying behaviors of simulation entities as a function of time. Frequently, the behavior of simulation entities is continuous in time. A large body of mathematics exists that provides us with structures like differential equations that implicitly specify the time-evolving behavior and are appropriate for such continuous behaviors.

On the other hand, a large body of literature and a number of languages (GPSS, Simula, Simscript) [BRATLEY, FOX and SCHRAGE 83] have been described that are suited to discrete-event simulation. In this work, a simulation is driven by events. The time at which an events happens in the future can be usually determined easily since the behavior taking place between events is usually quite simple. Events are kept in a sorted queue and the earliest event is activated at its precomputed time. The emphasis in discrete-event simulation is in gathering statistics about events over a time interval.

We are interested in simulations that are continuous in most part but various events may occur during a continuous behavior. These events cause the behavior of a system of objects to change and the system may start simulating according to

a possibly different behavior. The continuous behaviors of simulations are usually complex; the behavior can not usually be predicted a priori as to when an event will take place. The event occurs as a result of and as a part of the particular continuous behavior being simulated. The simulation of such systems needs a model in which the behavior rules that govern a system at various times, the events that may take place during each behavior and how an event triggers a new behavior are integrated into the same framework.

In this chapter, we present such a model. We discuss how time behaviors of systems of objects can be designed hierarchically as time behaviors during subintervals and connected to each other or plugged together through events. We start with the definition of an *event unit* that may be treated as a basic building block for the creation of time behaviors. We then present schemes to organize event units.

6.2 Classification of Time

We consider the following classification of time to be useful in a simulation environment.

Animation Time Animation time is the representation of time as it is appears in a motion sequence. This might be on film (normally 24 frames/second) or video tape (30 frames/second in NTSC). This time is what is *perceived* as "real time." Animation time is the rate at which that the results of a simulation will be presented.

Simulation Time This represents the time and the time scales at which a process is simulated. Most often, animation time differs from animation time by a scale and an offset. For example, in a simulation of galaxies, one second of animation time may represent millions of years of simulated time while in a simulation of atomic phenomena, one second of animation time may represent a few picoseconds.

Solution Time This represents the time elapsed during the solution of a problem. Solution time may be considered as the "real time" spent when a user is waiting for the computer(s) to produce solutions. It is difficult to predict the relation between solution time and simulation time or animation time. Widely varying amount of solution time may be required for a unit of simulation time depending on the computational complexity of the problem.

In each of the above cases, the behavior of a system of objects during a time interval will be partitioned into behaviors during subintervals. The sub-behaviors will take place in a sequence to generate the overall time behavior. The sub-behaviors may be

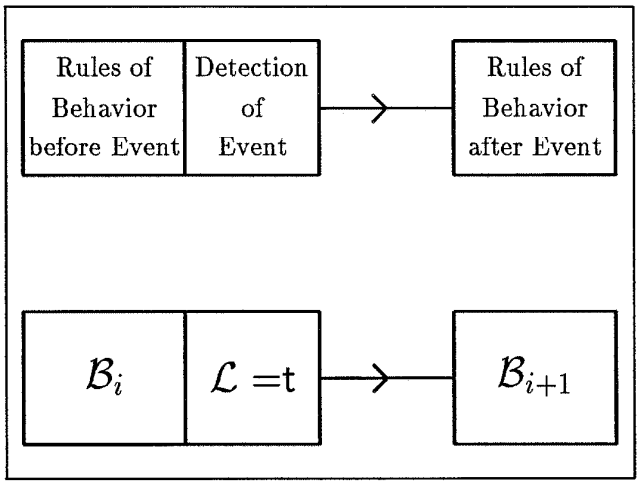

Figure 6.1: Representation of an "event unit." An event unit represents the local time behavior of a system of objects. The system initially simulates according to a behavior rule $\mathcal{B}_i$. When the logical function $\mathcal{L}$ becomes true, the system of objects switches to behavior rule $\mathcal{B}_{i+1}$.

complex and be simulated in different ways. For example, during one subinterval, the behavior of a system may be represented as a set of differential equations and during another subinterval, the behavior may be kinematically specified. The behaviors interact only through events. Various sequences of behaviors are possible depending on various events that may take place during a behavior. The exact sequencing will evolve as a result of the simulation and as a result of what events actually take place. The entire simulation may be set up as a "time program." The output of this program is the time behavior of the system during a time interval.

6.3 Systems of Events

Intuitively, an event signifies an "important" point in a simulation. Usually, there is a discontinuity associated with an event as in the case of a collision between two bodies. However, an object attaining a particular configuration or a variable attaining a particular value are also examples of an event although there may not be a visible discontinuity in the behavior of the system of objects. In addition, events have a time ordering. That is, a sequence of events can be arranged in a non-decreasing order in time.

Part of an event can be considered analogous to a conditional statement of a programming language. While a system of objects is simulating according to a set of behavior rules $\mathcal{B}_i$ and a logical condition becomes true, an event is said to have occurred. After the occurrence of an event, the system starts behaving according to another set of behavior rules $\mathcal{B}_{i+1}$ (Figure 6.1). In our representation, an event

happens at an instant of time and takes zero time, i.e., the simulation clock does not progress during the occurrence of an event.

6.3.1 Specification of an Event-System

An event is said to occur when a logical function $\mathcal{L}$ of the state $\mathcal{X}$ of a system of objects attains a true value. The event causes the system of objects to change its behavior from one set of rules $\mathcal{B}_i$ to another set of rules $\mathcal{B}_{i+1}$. $\mathcal{B}_i$, $\mathcal{B}_{i+1}$ and $\mathcal{L}$ together encode the local time behavior of a system and we call such a specification an *event unit.*

Hence an event unit $\mathcal{S}$ is specified as a triplet

$$\mathcal{S} : (\mathcal{B}_i(\mathcal{X}), \mathcal{L}(\mathcal{X}), \mathcal{B}_{i+1}(\mathcal{X}))$$

where, $\mathcal{X}$ is the state of the system of objects, $\mathcal{L}(\mathcal{X})$ is a logical condition signifying the event, $\mathcal{B}_i$ is the rules of behavior that the system is obeying before the event and $\mathcal{B}_{i+1}$ is the rules of behavior that the system obeys after the event. Clearly, the event signified by $\mathcal{L}$ happens during $\mathcal{B}_i$. A behavior $\mathcal{B}$ does not have to last for a finite time. Zero time behaviors set initial conditions for other behaviors which follow.

We find this break-down of the parts of an event useful to understand events. For example, when a collision takes place between two bodies, what is the event? Is it the instant of collision or is it the momentum transfer that takes place during collision? How does the motion of the bodies before and after the collision relate to the collision event? The separation of an event-unit as behavior rules before and after the occurrence of an event and a logical condition as the detector of the event gives a convenient division of the parts of an event.

Time Primitives: Building Blocks for Event-based Simulation Systems

An event unit as defined above can be treated as a building block for event-based simulation systems. *Systems of events* can be constructed by composing event units $\mathcal{S}$. Note that each of the behavior in an event unit can be composed of event units itself. Hence, event simulations can be built hierarchically.

Figure 6.2 shows an example of simple event units. Figure 6.3 shows an example of a zero length behavior.

6.4 Organization of Systems of Events

We defined an event unit in the previous section. An event unit is represented as a triplet

$$\mathcal{S} : (\mathcal{B}_i(\mathcal{X}), \mathcal{L}(\mathcal{X}), \mathcal{B}_{i+1}(\mathcal{X}))\,.$$

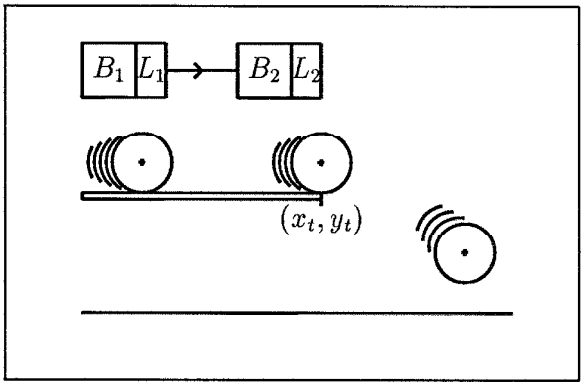

Figure 6.2: Simple event units. A ball is rolling off a horizontal plane. The ball is simulating according to behavior rule B_1, rolling on the table. An event L_1 happens when the ball reaches the end of the table. This event causes the ball to switch to simulate behavior rule B_2, a free fall under the force of gravity.

$\mathcal{X}$ represents the state of the system of objects. The system behaves according to behavior rules $\mathcal{B}_i$ until the event denoted by $\mathcal{L}$ takes place. The system then starts behaving according to the behavior rules of $\mathcal{B}_{i+1}$.

In this section we shall discuss composition of event units into more complex systems of events.

6.4.1 Initialization Behavior and Termination Event

In each system of objects, there is a default initial behavior and a default final event. The *Initialization* behavior is the behavior in which the system finds itself at the origin time of simulation time. This is a zero time behavior that initializes the state of the system. The *Termination* event is the event that causes the system to stop i.e., the program to terminate. We use the pictorial notation shown in figure 6.4 to indicate the termination event.

6.4.2 Composing two event-units

Two event-units

$$S_1 = (B_i, L_i, B_{i+1})$$

and

$$S_2 = (B_j, L_j, B_{j+1})$$

may be composed if

$$B_{i+1} = B_j$$

The composition could be thought of as resulting in a system of events

$$S_3 = (B, L_j, B_{j+1}),$$

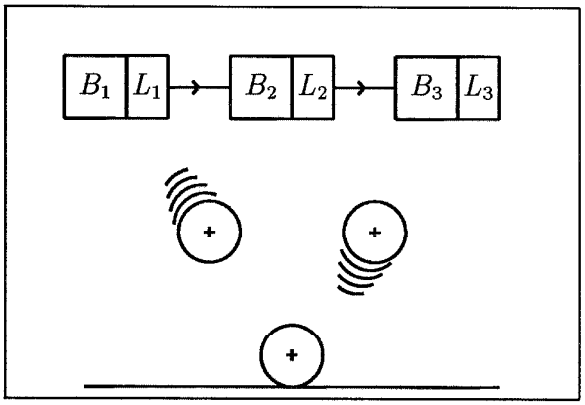

Figure 6.3: An system of events with a zero time behavior. A ball is free falling under gravity (Behavior rule B_1). An event L_1 happens when the ball hits the ground. The collision event causes the ball to go into a zero length behavior B_2 in which momentum transfer computations are made. An event L_2 is caused after the momentum calculations and the ball goes into a free fall behavior B_3. The momentum transfer behavior lasts for zero time although it causes a change in the behavior of the system of objects.

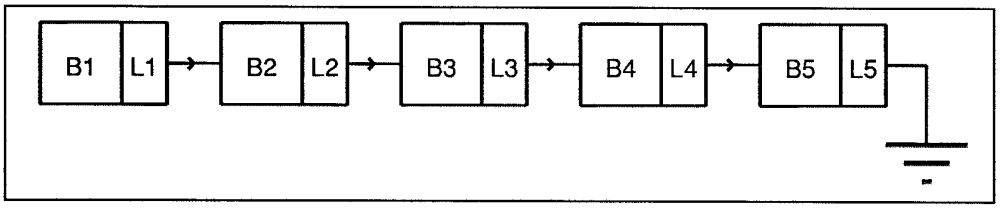

Figure 6.4: A time line representation. The system of objects simulates in behavior rule B_1 until event denoted by L_1 takes place. The system then switches to simulate with behavior rule B_2 and so on. After event L_5, the system stops simulating. The "Ground" symbol is used to denote the termination event system(when the simulation stops).

where B represent a behavior rule composed of (B_i, L_i, B_j). L_j is the event detection function and B_{j+1} is the new behavior rule for S_3.

This simple composition can be extended in several ways to give some useful time sequences. In the following sections, we progress from simple to more complex organizations of event units.

6.4.3 Time Graphs

Systems of events can be connected to each other using a directed graph with event systems being the nodes and edges representing the connection between the event units. In this subsection we present some special cases of this general connection network.

Time Lines

The simplest organization of event systems is in a linear arrangement which is monotonic in time (figure 6.4). We call this organization, a *timeline.* In a time line, each system of events can be entered from only one other event system, only one event can happen in that event system and the event system can lead to only one other event system. Formally, a time line is composed of event systems as

$$L = \{ \begin{array}{ccc} (B_0, & L_0, & B_1) \\ (B_1, & L_1, & B_2) \\ \vdots & \vdots & \vdots \\ (B_{n-1}, & L_{n-1}, & B_n) \} \end{array}$$

Time lines are quite useful; many computer simulations with events have been implemented with time lines.

An example of a time line is shown in figure 6.5.

Time trees

More than one event may occur in a behavior rule causing the system of objefts to go into one of several new behavior rules (figure 6.6). A *time tree* is an extension of time lines that allows such multiple connections between event units. In a time tree, associated with each event unit is set of event units, representing the new behavior rules for the system of objects, with possibly more than one event unit in the set. The system of objects simulates in its current behavior until any of the possible events occurs, after which the system migrates to the event unit associated with the event which occurred. A time tree is specified as

$$T = \{ \begin{array}{l} (B_1, \{(L_{11}, B_{11}), (L_{12}, B_{12}), ...\}) \\ (B_2, \{(L_{21}, B_{21}), (L_{22}, B_{22}), ...\}) \\ \vdots \\ (B_n, \{(L_{n1}, B_{n1}), (L_{n2}, B_{n2}), ...\}) \ \} \end{array}$$

In this representation, each behavior rule B_i has a set of (event, next behavior), (L_{ij}, B_{ij}) pairs associated with it. While the system of objects is simulating in the behavior rule B_i, any of the events L_{ij} can occur. The event Lij occurring in B_i will cause the system to shift to behavior rule Bij.

A pictorial representation is shown in figure 6.6.

Unlike time lines, time trees can represent multi-way branches in a simulation. Although there is a *possibility* of more than one event happening at a node, only one path is chosen during simulation. That is, simulation progresses along a line from the beginning to the end traced in the time tree as shown by the bold line in figure 6.6. A time tree is analogous to a "switch" statement in the C programming language [KERNIGHAN and RITCHIE 78], when more than one outcome is possible although only one is chosen.

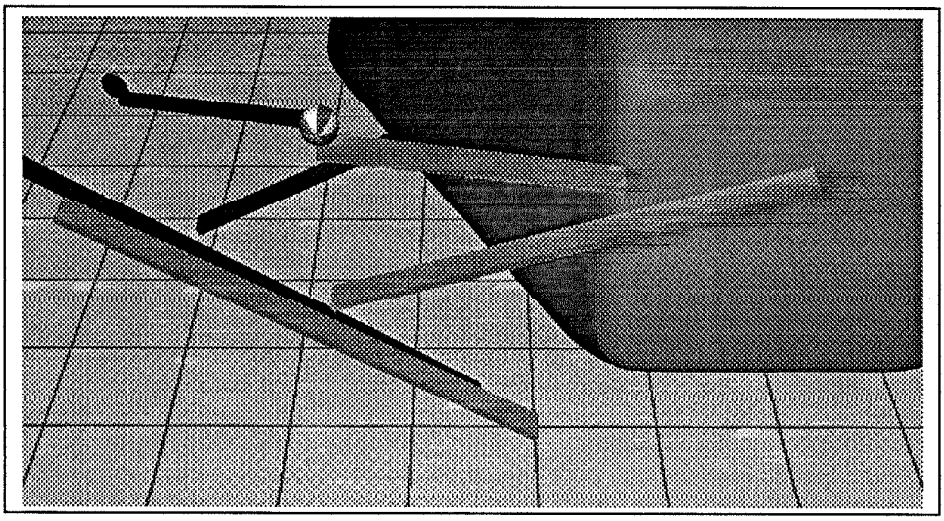

Figure 6.5: A simulation generated from a time line organization of events. A ball rolls down three incline planes A, B, C. At each incline plane, the only event that can happen in this simulation is reaching the end of the plane. This event causes the ball to start free falling under gravity. During free fall, the only event that can happen is hitting an incline plane. This event takes it into incline roll behavior.

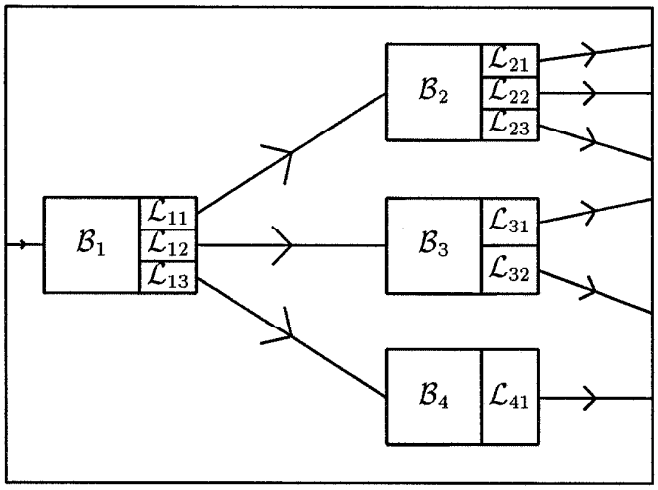

Figure 6.6: A Time Tree organization of systems of events. Three events may happen when the system of objects is simulating with behavior rule B1. The system of objects migrates to behavior rule B2, B3, B4 depending on whether event L11, L12 or L13 happens respectively. Also a behavior rule may be reached from more than one system of events as behavior B2 in this figure. The actual path chosen by the system of objects is shown as the bold line.

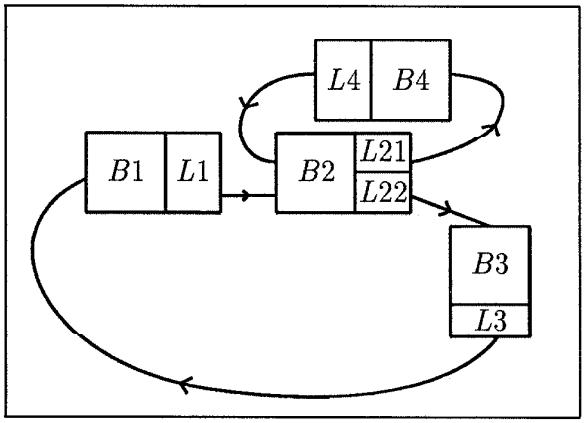

Figure 6.7: An Event Graph. An event graph $\mathcal{G}$ is a general organization of event units. The nodes are event units and the edges represent the connections between event units. Event graphs may contain loops.

Event Graphs

Time lines and time trees do not contain any cycles. The simulation system goes through each behavior at most once. In *event graphs*, a system of object can visit an event system that it has been to before (Figure 6.7). A time tree is a special case of event graph with no loops. A time line is a special case of time tree, with one outgoing edge per node. An event graph is a connected, directed graph.

An example of event graph is shown in the simulation set up in figure 6.8. This simulation contains both multi-way branches and loops. The simulation was created by programming separate continuous behaviors and connecting them together at various events. Note that we can mix different kinds of behaviors in simulations like these. One part of the simulation may be simulated as a set of differential equations while another may be specified kinematically. The behaviors only interact at the events.

Further, once the time graph has been configured, various behaviors of the system of objects can be observed under different conditions. In the simulation sequence of this Rube Goldberg machine, when the ball B reaches the height of the piston P, it can either get knocked off of the conveyor belt or else continue upwards. In another run of the simulation, we can switch off the piston and the system simulates correctly interacting with other objects in the environment. We could similarly change the radius of the ball, slopes of inclines, rising speed of the elevator etc., and the system would automatically generate the right behavior. We have in effect created a time program to which we can now feed different input data. We could even remove parts of the environment such as one of the incline planes and the simulation would stay correct. This is equivalent to editing our time program.

All the event unit organizations described above, time lines, time trees and time graphs can be themselves used as behavior rules in an event unit.

With the constructs described in this section, we have defined primitives, conditionals, multi-way switches and loops.

6.4.4 Merits of Time Primitive Abstraction

The above abstraction of event units and their organization provides the following advantages:

- Behavior rules can be programmed individually for each event unit. The interaction with the rest of the system of objects is through events and event transitions. This provides a partitioning of the time behavior design of a simulation.

- A behavior function can be designed as if the function starts simulating at zero time, that is, we can use a canonical time coordinate system for all behaviors. A behavior is triggered by an event and the time of the event is the time offset for the behavior. A simulation engine can maintain the correct time offsets for each subsystem of objects.

- Systems of events can be hierarchically composed to create more complex event systems.

- The organization of event systems gives us programming constructs for describing motion sequences. We can apply good programming paradigms to construct reusable behaviors and easily modifiable event sequences. We can also create a library of behavior rules that can be connected together in various ways.

- The graph representation of event systems gives us the opportunity to use the well developed concepts of graph theory [DEO 74] to analyze event systems.

6.5 Applicability of Time-Event Approach

The examples in section 6.4, which illustrate the event-time approach, have been selected from computer animation. The time-event approach presented above can also used in interesting ways in other domains. Some of the applications are presented below that are suggested by the definition of an event-unit presented in section 6.3 as events causing system of objects to switch from one set of behavior rules to another set. A big advantage of this approach, as stated before, is that various behaviors may be programmed separately and can be plugged together. The behaviors may be any time-dependent phenomenon.

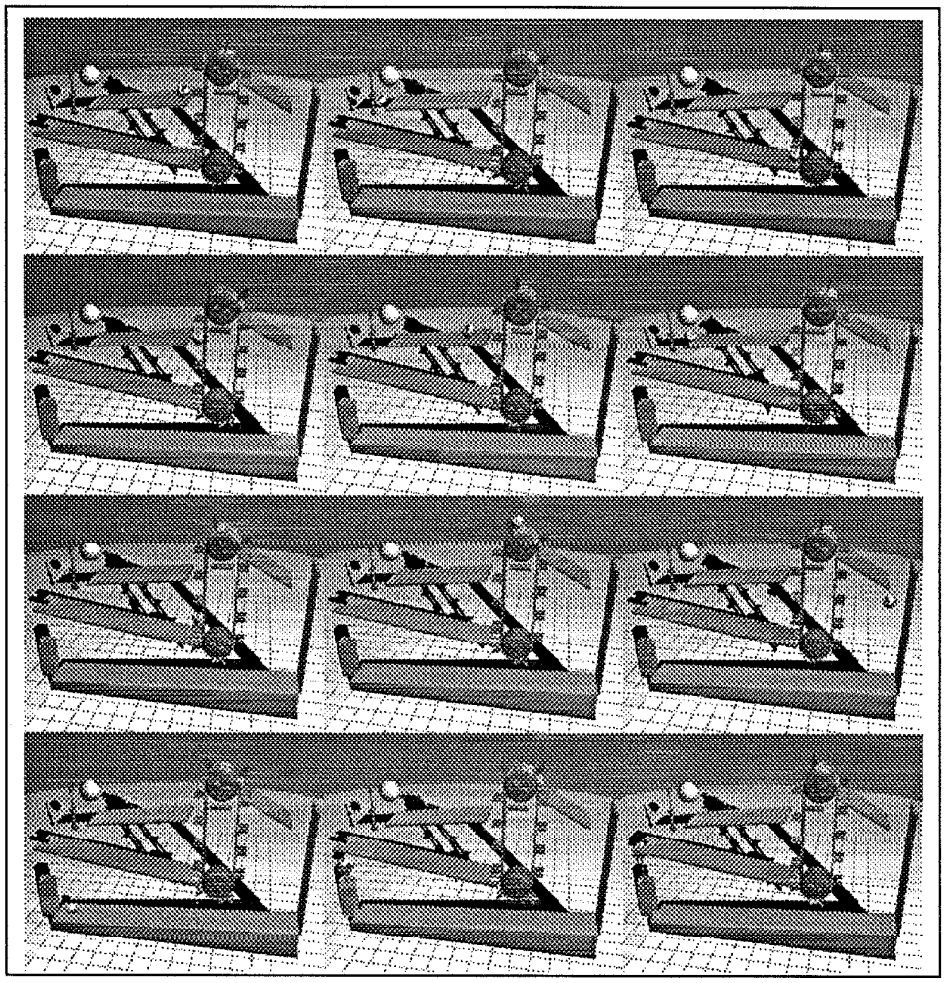

Figure 6.8: A simulation generated from an event graph organization of events. The graph for this simulation contains multi-way branches and loops. The graph has 26 event units and two loops. A multi-way branch takes place when the ball is knocked by the piston on to plane A or travels upwards if not knocked off.

Changing problem solution

A solution strategy to solve a problem can be cast in a time-event approach. Some examples of events occurring during a solution process are some variable taking a value outside prescribed limits or an iterative solution having taken too many iterations. The new behavior triggered by these events could be a more robust solution technique. For example, in the simulation of a bending beam, different sets of equations may govern the behavior of the beam depending on the amount of bending. A simple linear elasticity model may be used for small bending but a more complex model may be needed to accurately model large bending behavior.

Changing Object representation

The representation of model used for an object may also be changed depending on various events happening in a system of objects. For example, an object may switch its representation from a collection of particles to a fluid to a rigid body depending on various events such as temperature change. Similarly, a large collection of objects may be simulated as a continuum or even a single particle if, for example, the projection of the collection of objects on the image plane is a small fraction of the full image. [BARZEL 91] is exploring such variable representations of objects. Our work presents a way to use these representations and to switch between them.

6.6 Implementation of Systems of Events

In this section, we discuss the implementation details of a simulation environment for events with the abstraction presented in this chapter.

We assume that during the simulation using event-systems, the results of the simulation are sampled at predetermined times. We require that the system of objects being simulated be in the correct state at the time the results are sampled. In the case of making an animation, this snapshot is taken at every frame time. In simulating a system of objects , we have to reconcile all the events happening asynchronously to the frame clock and simulate the system across all the events in the order of occurrence of the events to obtain the correct state at the frame time.

6.6.1 Simulating a System of Events

The basic event simulating loop for a system of objects S is:

1. Simulate system S in current behavior until an event happens
2. Switch system S into the behavior corresponding to the event that happened

There is an implementation detail, however, that has to be taken care of. During the simulation of a system of objects, most events are detected some time after they

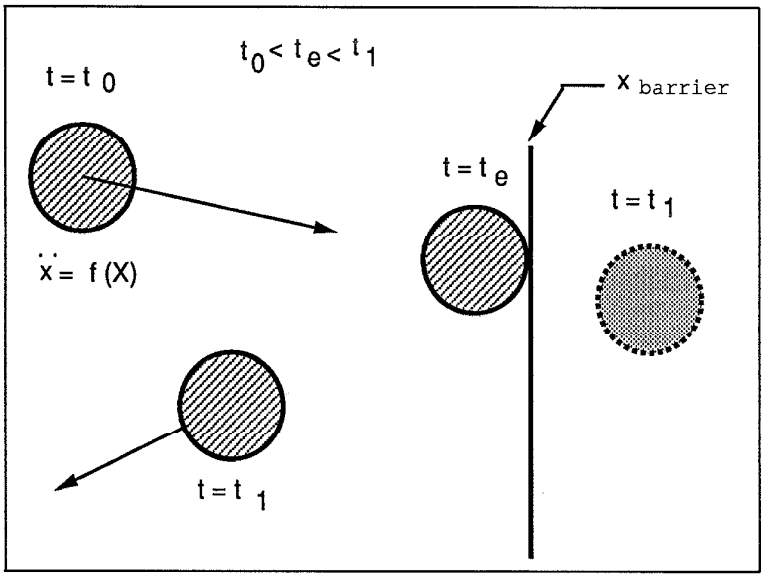

Figure 6.9: The need for a method to compute time of occurrence of events in a simulation using discrete sampling times. Integrating the continuous equation of motion of the particle between discrete sampling t_0 and t_1 times does not take into account the collision event at t_e. The event has to be detected, accounted for, and the simulation restarted to get the correct behavior.

have taken place. For example, consider the example of a particle moving towards a barrier at $x = x_{\text{barrier}}$ (figure 6.9) following the equation of motion:

$$\frac{dx}{dt} = f(\chi)$$

The motion of the particle is found by integrating this differential equation. The event of the particle hitting the barrier is given by

$$x_{\text{particle}} = x_{\text{barrier}}$$

In general, we cannot solve for the time of this collision event in closed form. We can detect that collision has happened by noticing that

$$x_{\text{particle}} > x_{\text{barrier}}.$$

If the differential equation is being integrated in discrete time steps, the collision detection may happen some time after the time of the collision. The exact time of the collision has to be computed to be able to handle the collision correctly. To restart the simulation at the time of the event, the state of the system of objects just before the event has to be determined. Starting from this state, the system switches into the behavior dictated by the event that took place.

We have provided two mechanisms to implement this backtracking for the state of a system of objects to the time of an event. The first mechanism is general. This mechanism saves the state of the system before simulation is started for any time interval. If an event takes place during that time interval, the system can be restarted with the saved state and resimulated until the time of the event. The second mechanism is provided for efficiency but is less general. This step may use interpolation of the state of the system back to the time of event from the time the event is detected.

Simulating an event-system

Each behavior rule $\mathcal{B}_i$ is programmed to simulate the system of objects S to a specified time t_{n+1} from a known state at time t_n, unless an event occurs before time t_{n+1}. If any events happen before time t_{n+1}, the behavior rule returns the time of each event that occurred. The time t_{n+1} is the time at which the results of the simulation will be next sampled. If no event is reported by the behavior rule $\mathcal{B}_i$, it implies that the state of the system S has now been updated to time t_{n+1}. If one or more events are reported, then the system S needs to be restarted at the time of the earliest event. From all the events reported by $\mathcal{B}_i$, let t_e be the time of the earliest event. Then state of the system S is computed at t_e just before the event. The simulation of system S is now started from time t_e in the new behavior rule $\mathcal{B}_{i+1}$ dictated by the event that occurred.

A behavior rule may return more than event. However, only the earliest event is important. Therefore, if it is efficient, a behavior function may report only one event if it is guaranteed that the reported event is indeed the earliest one.

The main simulation skeleton to simulate an event-system is shown in pseudo-code in figure 6.10.

The implementation described is independent of the frequency of occurrence of events. This of course assumes that each behavior rule $\mathcal{B}_i$ is capable of detecting any event that can occur when the system is behaving according to the behavior rule $\mathcal{B}_i$. The time-event approach adapts to "time stiffness" of the problem. It simulates slower during time duration where more events are happening and faster where fewer events are happening. The correct state of the system of objects is obtained at the time the state is sampled irrespective of how many event units (events and behaviors) that the system has to simulate through. During the generation of the Rube Goldberg machine simulation shown in figure 6.8, we could simulate with any amount of time between two frames, and any number of events occurring between two frames, and we would always get the same state at the same animation time.

6.7 Summary

In this chapter we presented a methodology to partition behaviors of systems of objects in time. We presented a time primitive that we call an event-unit. Each event-unit represents a current behavior, events that can take place during the current behavior and the next behavior for the system due to each event. Event-units can be organized as a directed graph to represent the time behavior of a system over an interval of time in terms of time behavior of the system during subintervals.

```
/*
       Simulation of a system of objects S
    t_n :      time at which the state of S is known
    t_(n+1) : time at which the state of S is desired
    t_s :      time to which S has been simulated
        t_s = t_n at the beginning of the code
        t_s = t_(n+1) at the end of the code
*/

while(t_s < t_(n+1))
    Save state of S

    Simulate S in current behavior B_i
      until t_(n+1)  (unless an event occurs before t_(n+1))

    If any event occurred
        find time t_e of the earliest event E_e
        compute state of S at time t_e
        t_s = t_e
        put S in behavior B_(i+1) corresponding to event E_e
    else          /* state of S has been updated by B_i to
                     time t_(n+1) */
        t_s = t_(n+1)
    endif
endwhile
```

Figure 6.10: Pseudo code for simulation of a system of objects S. The system state is known at t_n and is desired at t_{n+1}. System S is simulated in current behavior $\mathcal{B}_i$ until an event is detected. The state of system S is computed just before the event and the system is switched to the behavior $\mathcal{B}_{i+1}$ indicated by the event that occurred.

Part III

Language, Implementation and Examples

Chapter 7

Language and Implementation

In previous chapters, we have presented the concepts for the design of a unified constraint-based simulation system. As we stated, the primary goal of our design is the ability to use multiple techniques in the same framework. As discussed in chapters 3 through 6, we decided to identify basic primitives for a simulation system and to build on them hierarchically.

Implementation of a Constraint Environment

We now wish to implement a constraint environment based on the concepts presented in the thesis.

The constraint environment needs to be programmable so that the various primitive blocks can be put together to create powerful simulation approaches. We need a good programming paradigm to provide the ability for object-oriented programming and data abstraction. The environment should be portable across various machines and operating systems. The environment should be extensible both by the users and the designers. The system should provide functional transparency, i.e., the interface language for the system should be similar to the language used to construct the system so that new user-defined routines can be integrated into the system smoothly and the users can use the system routines with ease.

We have decided that the basic interface to the environment is through a general programming language: an extension to C++. The substrate of the system is designed in the same language. We have also implemented some basic building blocks in the substrate to create a prototype environment system called **O**ur **C**onstraint **E**nvironment (OCE). OCE has been used to create simulation examples which are presented in the next chapter.

7.1 Design of an Interface Language

There are a number of possible approaches for designing a language to interface to a simulation system, each having its advantages and disadvantages. We present some standard choices that are available to anyone designing a language for a simulation system and then we will discuss our choice.

7.1.1 Possible Approaches for Interface Language Design

Choice A: Designing a New Specialized language

This approach involves designing a new language which is specialized for one particular purpose. Some robotics languages have been designed using this approach [SHIMANO, GESCHKE and SPALDING 84][GRUVER and SOROKA 88]. A robotics language will usually have constructs to specify the motion of a robot. Additional constructs may be added, such as to describing sequences of motion and executing conditional branching. The advantage of designing a specialized language is that it is often possible to design elegant structures for the specific purpose for which the language is designed.

However, in a practical application, there are a number of peripheral tasks that need to be performed. In the robotics example, matters such as operator interface, calibration procedures and calculations, error recovery logic and input date processing have to be taken care of. In fact a major part of an application may require constructs available in general purpose programming languages [TAYLOR, SUMMERS and MEYER 82]. Further a specialized language approach does not typically lend itself well to extensibility since the language is, by design, good at specific functions.

Choice B: Extending an Existing Programming Language

This approach to the design of an interface language starts with an existing general-purpose computer language. This language is then extended by adding necessary syntax and semantics for the specific purposes of the application to which the language interfaces.

The advantages of this approach include, 1)the basic programming language has already been designed, 2) many design questions for the language design have already been answered, 3) a large amount of user documentation and user training exists, and in many cases 4) an implementation of the language exists.

The disadvantage of this approach is that the designer of almost any language, even a general-purpose language, will have made some compromises. This might make the base language restrictive in some ways that may be detrimental for the application to which the extended language interfaces.

Choice C: Writing Subroutine Libraries in a General Purpose Language

In this approach, the simulation system is written as subroutine library in a general purpose language. Various subroutines are provided to do some simulation-specific

functions. To create a simulation, a user writes a program in the same language that was used for writing the simulation system. The program is mostly constructed with subroutine calls to the subroutine library and control structures from the base language.

This is a straight forward-approach and one that is fairly easy to implement. However, the user is limited by the decisions taken by the subroutine designer. The user has to adhere to the interfaces provided by the designer. The user is often unable to change the internals of the system in any way, since the only interface to the system is provided through subroutine calls.

Choice D: Designing a General Purpose Language

This approach involves designing a new general purpose language as a programming base and then extending it with functions necessary for the specific application for which the language is an interface. This is the most flexible approach. The base language can be designed with exactly the features that are considered appropriate for the application at hand without the constraints of an existing implementation.

This approach also involves the most effort since the design and implementation are done completely bottom up. It is also easy to fall into the trap of "creeping featurism." There is a tendency to add new, often ad hoc, constructs to the base language as required by the application. A user is also forced to learn the syntax of a new language in addition to application-specific information.

7.1.2 Language Design Approach in OCE

We selected choice B, selecting an appropriate existing programming language, and extending the language both in syntax and semantics.

Considering the design requirements and philosophy presented in this thesis, we found this approach to be the most attractive. We wanted a powerful language to write a substantial substrate. We also wanted the capability to extend our system by migrating applications as they are developed into the substrate. For this capability, it was preferable to use the same language for the interface and the substrate. There are some simulation-specific constructs and facilities that are not generally present in a general-purpose computer language. We decided to add these syntactical and semantic constructs to the language we have chosen, C++[LIPPMAN 89]. Using this approach we quickly built a prototype simulation system, since implementations of the base language already existed.

7.1.3 Choice of a Base Language

We need our base language to have some basic capabilities: the language needs to provide a good programming paradigm with data abstraction, structured programming and object-oriented programming capabilities. The language must be easily

extensible such as providing capabilities to define complex user-defined types. We require the language to be fairly efficient, since intensive numerical computations form a major part of typical simulations. To provide an interactive but efficient interface, we want the capability to have the language run in both interpreted and compiled modes with the interpreter being able to interface to compiled code. A good support environment with powerful compilers and debuggers is also desired. Portability to a number of machines is also a requirement.

There are number of languages that provide some of the above features to varying degrees. Smalltalk [GOLDBERG and ROBSON 83] provides a good programming model and a good environment but most Smalltalk implementations are slow and Smalltalk is not widely available. Lisp, particularly with object oriented extensions as in Flavors, is becoming widely available but is not sufficiently efficient even on platforms designed specifically to run Lisp. Most procedural languages like Fortran, Pascal and C are efficient and widely available but provide a low level programming model.

As a compromise between our various requirements, we chose a recent language called C++ (read as C plus plus). C++ has a syntax similar to the programming language C. However, C++ offers facilities to program in an object-oriented manner with capabilities for data-abstraction. Most C++ implementations translate C++ code into C code which is then compiled and the run-time code is still efficient. Although C++ was released only about 5 years ago, it is already available on a large number of computing platforms, from main-frames to personal computers. Optimizing compilers and powerful debuggers are available on most of these platforms.

7.2 OCE Implementation

The implementation of OCE represents a computational formalism for the concepts explained in the previous chapters. The current state of OCE represents a prototype to test the concepts provided in this thesis. The description provided below is intended to provide ideas about different approaches, methods and techniques that may be implemented in a unified constraint environment. The current implementation, albeit supporting a limited number of objects, has been used to create a number of simulations. These simulations will be presented in the next chapter.

7.2.1 Review of C++: Existent Features of the Base Language

The substrate of OCE is written in C++; the interface to OCE is our extension to C++. C++ was developed as an extension to the C programming language. The C programming language is very efficient, but it provides a very low level computing model. C++ was designed to provide support for data abstraction and object-oriented programming with type checking. Most of the constructs in C++ are similar to the C

programming language. Some of the additional facilities in C++ are described in this section.

Support for Data Abstraction

C++ provides support for user-defined types, called, *classes*, that can be used as conveniently as built-in types. A class contains both the declaration of data and member functions or "messages" that operate on the data. The semantics match the definition of an object we presented in chapter 2. For example, a complex number class may be defined by a user as:

```
class complex{
    double re, im;    /* Real and Imaginary parts */
public:
    complex operator+(complex);  /* addition of two complex numbers */
                      /*    Definition of function appears elsewhere */
};
```

The data declared in a class may be `private`, `protected`, or `public` which makes the data available to objects of that class, objects of derived classes or objects of any class respectively.

With this declaration, two complex numbers may be added with the same syntax as the syntax used to add two integers:

```
complex a, b, c;
    c = a + b;
```

A clean interface to an object can be created by encapsulating both the state data of an object and all the functions that operate on the state data into the definition of the object. The data type can then be used as a black box knowing only the properties of the object type without having to know all the implementation details.

Support for Object-Oriented Programming

C++ offers the ability to program in an object-oriented manner while still providing reasonable efficiency. Generic classes for a category of objects may be defined. Specific instances may then be "derived" or "inherited" from these generic classes. The derived class inherits all the data and functions declared in the class from its parent class. In addition new data and member functions may be added. There also exists a mechanism to override the functions defined in parent classes. For example, a general class, `geometric_shape` may be defined as:

```
class geometric_shape{
    point center;
public:
```

```
    virtual void draw();
    virtual void rotate();
};
```

A `virtual` function is a member function in a class that can be overloaded by a function of the same name in a derived class. The keyword `void` indicates the type of the result returned from the respective function, i.e., null or no result is returned through the name of the function.

Specific classes like rectangle, circle and line may then be derived from this general class as:

```
class circle : public geometric_shape{
    double radius;
public:
    void draw();
    void rotate();
};
```

The `draw()` and `rotate()` functions of class `circle` override the corresponding functions provided in `geometric_shape`. The class `circle` also adds the data for `radius` of the circle to the data inherited from its parent(s), in this case, `center`.

Note that the syntax in C++ to call a member function `f(<arguments>)` of an object `obj` is

```
obj.f(<arguments>)
```

In this way, the common aspects of various objects can be abstracted by using inheritance.

C++ allows calls of member functions depending on the type of the object, even if two functions have the same syntactical representation. For example, if we define a vector class as:

```
class vector{
    double x[3];
public:
    vector operator+(vector);
};
```

the vector addition function is called for `a+b` if `a` and `b` are of type vector and a complex number addition function is called if `a` and `b` are of type complex.

Choice of Function Names According to Functionality

More than one function in C++ may have the same name. The appropriate function is selected depending on the type of object of which the function is a member, and depends on the type of parameters of the function. This implies that functions performing similar actions on different types may be called by the same (hopefully intuitive) name rather than requiring several awkward made-up names. For example, a function that makes the image of an object on the screen may be called `render()` for any type of object. Then the statement `body.render()` will automatically execute the render function appropriate to the type of the object `body`.

One corollary of using parameter types to select the appropriate function to be called is that C++ provides strict type checking, lack of which was a common source of errors in C.

7.2.2 Implementation of Refinement Layers in OCE

Now that we have reviewed C++, we proceed to describe the implementation of the refinement layers presented in chapter 4. Unlike chapter 4, we will describe the implementation starting from the bottom-most layers to the higher layers. This is the same order in which OCE was implemented.

Numerical Solution Techniques

We consider numerical solution techniques as the "indivisible" primitives of OCE. We have provided our extension to C++ as the base language to OCE; all the control structures and facilities of a general purpose language are available. A constraint environment needs to provide a large set of numerical techniques so that a user of the constraint environment is able to construct simulations by putting together building blocks, rather than having to write a large amount of code in the base language.

As a beginning of a powerful and extensible set of numerical solution techniques, we chose NAG Version 13 [NAG 89], an extensive library of numerical routines. NAG is written in Fortran 77. We considered it important to provide a similar interface in each layer of the constraint environment. Since the OCE base language is extended C++, we wrote a C++ interface to every routine of NAG[1]. Since the calling conventions of routines in Fortran are different in C++, our C++ interface removes the need to use a different interface than the rest of the system. C++ also provides argument type-checking during subroutine call. Through the C++ interface we generated, we now have type-safe usage of an extensive fortran library without the need of a fortran run-time environment.

The details of a generic numerical interface will, of course, depend on the language and implementation of the numerical techniques. For the specific example of providing

[1]The interface was created automatically from NAG's online documentation in collaboration with Ronen Barzel at Caltech.

an interface to NAG, a fortran library, we needed to reconcile the calling conventions of fortran with that of C++. A fortran subroutine is called with all of its arguments passed by reference, i.e., the address to the data is passed to the routine. C++ provides facilities to take the address of a data item and we could have written our interface using these facilities. For example, a routine in NAG which has a fortran interface as:

```
      SUBROUTINE C05ADF(A, B, EPS, ETA, F, X, IFAIL)
C     INTEGER IFAIL
C     real A, B, EPS, ETA, F, X
C     EXTERNAL F
```

could have a C++ interface as

```
void c05adf(double *a, double *b, double *eps, double eta,
            double (*f)(), double *x, int *ifail)
```

The above interface to the routine `c05adf` requires that the address of each argument be passed and would be called as:

```
double a, b, eps, eta, x, f();
int ifail;
 :     :    :
c05adf(&a, &b, &eps, &eta, f, &x, &ifail);
```

We have implemented a cleaner interface by using the ability in C++ to pass arguments by reference. Additionally, if an argument to a procedure is not changed in the procedure, the argument can be declared as a constant. Using the syntax for passing arguments by reference and declaring the argument as constant if necessary, the interface to `c05adf` is:

```
void c05adf(const double &a, const double &b, const double &eps,
            const double &eta, double ( const *&f)(), double &x,
            int &ifail)
```

and a call would be simply:

```
double a, b, eps, eta, x, f();
int ifail;
 :     :    :
c05adf(a, b, eps, eta, f, x, ifail);
```

Note that this second implementation of the interface generates a cleaner call syntax than the first implementation. In general, the translation of the fortran interface to the C++ interface is shown in figure 7.1.

Fortran Parameter	C++ parameter
INTEGER X	int &x
INTEGER X(N)	int x[/* N */]
real X	double &x
real X(N)	double x[/* N */]
LOGICAL X	char &x
SUBROUTINE X ...	const void (*&x)(<arguments>)
INTEGER FUNCTION X ...	const int (*&x)(<arguments>)
real FUNCTION X ...	const double (*&x)(<arguments>)

Figure 7.1: Translation of NAG fortran interface to a C++ interface for OCE. If an argument is unchanged in a routine, const is prefixed in the C++ declaration. The term 'real' is used to represent the implementation for floating point numbers, 'float' or 'double precision'. In the implementation of NAG on our computer system, 'real' is implemented as double precision.

Some of the basic categories of routines available in NAG and hence in OCE include roots of polynomials and transcendental equations, quadrature, ordinary and partial differential equations, optimization, matrix operations, eigenvalues and eigenvectors, simultaneous linear equations, and curve and surface fitting.

Symbolic Manipulation and Data Structuring

The symbolic manipulation and data structuring layer is implemented to present two kinds of support. First, the layer provides support to generate code and data structures that are not directly related to the central mathematical problem being solved but are needed for a specific numerical solution technique. An example is computing the gradient of a function to be optimized. Second, the arguments to the primitive numerical solution techniques are basic data types provided in the base language. To exploit the data abstraction facilities of C++, we would like to be able to use user defined types as arguments to solution methods. The layer also provides this data structuring support.

We have written a stand-alone experimental interface to the symbolic manipulation package, *Mathematica* [MATHEMATICA 88], that can be used to generate C++ routines from *Mathematica* output. This interface can start up Mathematica as a coprocess and communicate with the package using UNIX sockets. The interface currently understands a limited syntax that lets us generate some mathematical functions symbolically such as differentiation and integration. Using this interface, we have written a package to generate the inertia tensors for a given mass distribution. The interface can be also used to compute some auxiliary structures such as gradients of functions to be used in optimization or other problems. This interface will be incorporated into OCE.

Data Structuring provides support to use a data type that can be represented in multiple ways with a uniform interface. In the current implementation, we have implemented some classes of data items that are useful in simulation problems. The emphasis in the implementation of these classes was to provide an interface compatible to the use of the data item but independent of the representation of the data item. Currently, the classes implemented in the data structuring layer in OCE are vectors, matrices and rotations.

Vectors

A vector of size n may be declared as:

```
Vector v(n)
```

The following operations on vectors are available:

```
vector v1, v2, v3;
matrix m1;
double d1, d2;
v3 = v1 + v2          /* vector addition */
v3 = v1 - v2          /* vector subtraction */
v3 = d1 * v1          /* multiply with a scalar */
d1 = v1 . v2          /* dot product */
v3 = v1 ^ v2          /* cross product */
Matrix m1 = v1 | v2   /* outer product */
Matrix m1 = v1.dual_matrix() /* See appendix A */
v3 = v1.unit_vector()
d1 = v1.magnitude()
```

Three component vectors have been implemented as a special case for efficiency. Three component vectors may be used as the general vectors above and are declared as:

```
Vector3 v;
```

Matrices

Support for arbitrary-size matrices is provided. In addition sparse matrices are implemented with the same interface as other matrices. In particular, a general sparse matrix, banded matrices and triangular matrices have been implemented. Each of the types of matrices is derived from the class `GenericMatrix`. The different types of matrices may be declared as follows:

```
Matrix m(n, m);         /* Matrix of dimensions n, m */
SparseMatrix sm(n, m);  /* Sparse matrix of dimensions n, m */
```

```
DiagMatrix dm(n, m, b); /* Diagonal matrix of dimensions */
                        /* n, m and bandwidth b */
TriangularMatrix tm(n, m, {UPPER|LOWER});
                        /* Triangular matrix of order n, m. */
                        /* flag indicates upper or lower */
                        /* triangular                    */
```

After a matrix (of any type) is declared, it can be accessed and operated with uniformly.

Similar to vectors, addition, subtraction and multiplication of matrices with scalars, vectors and matrices exist. All multiplications uniformly use the '*' operator as:

```
double d;
vector v;
matrix m1, m2, m3;
m3 = d * m1;
m3 = m1 * m2;
m3 = m1 * v;
```

Additionally, operations for transpose, inverse, adjoint, column and row extraction of a matrix exist.

Rotations

A general representation for rotations has been defined. Rotations are internally represented as quaternions (Appendix A). However, programs can interface to the internal rotation representation through other representations of orientation, such as a rotation matrix or **euler-angles**. Transformation of vectors and second order tensors (such as Moments of Inertia) by rotation matrices is available using the same syntax irrespective of the representation of rotation.

A rotation object may be created using a matrix, axis angle, quaternion or euler angle representations. In addition, there are several choices of Euler angle representations. We chose the *roll, pitch* and *yaw* or **xyz** euler angle representation. A rotation object may be created as:

```
rotation r(matrix m)                                  /* rotation matrix */
rotation r(vector axis, double angle)                 /* axis, angle     */
rotation r(quaternion q)                              /* quaternion      */
rotation r(double roll, double pitch, double yaw)/* euler angle     */
```

Generic Numerical Methods

The routines implementing primitive numerical solution techniques need a large number of arguments that are very specific to the numerical technique being used. The

purpose of the generic numerical methods is to provide an interface for the solution of general mathematical problems. The layer attempts to reduce the need to account for all the details of primitive numerical solution techniques that will be used to solve the generic mathematical problem. In each of the generic numerical method implementations, a numerical solution technique is used as a default technique. A numerical solution technique other than the default may be specified by the user. OCE currently provides solution of the following generic mathematical problems.

Interface to Linear Equations

This interface provides a solution to a set of possibly non-square set of linear equations. The class of problems accepted are:

Find vector $\mathbf{x}$ such that

$$\mathbf{Ax} = \mathbf{b}$$

where

$\mathbf{A}$ = an $m \times n$ real matrix
$\mathbf{x}$ = vector of n unknowns
$\mathbf{b}$ = a vector of m constants
m = number of equations in the set, possibly equal to n

If the system is non-square, a **minimal least squares solution** is found. A method to take advantage of the sparsity is used if the matrix provided to the interface is sparse. If the matrix does not use representation for sparse matrices but the user specifies that a sparse matrix be assumed, the sparse matrix interface provided in the data structuring layer is used to transform the matrix into the proper data structure before calling the numerical routine from the numerical method layer.

The matrix $\mathbf{A}$ can have any of the representations, full, sparse or banded, as described in the previous section. By using the data structuring facilities provided for matrices irrespective of internal representation, the same interface can be used to the linear system solution for any matrix type. Further, efficient solutions can be chosen for sparse matrices automatically.

The uniform solution interface is:

```
solve_linear_system(GenericMatrix m, Vector x, Vector b)
```

The default method used for the solution of linear systems is **LU-Decomposition**. Particular solution methods may again be chosen by calling the routine as:

```
solve_linear_system(OCE_choice method, matrix m, vector x, vector b)
```

Interface to Ordinary Differential Equations

The interface to the solution of ordinary differential equations provides a generic interface to the solution of the system:

$$\begin{aligned} y_1' &= f_1(y_1, y_2, \ldots, y_n, x) \\ y_2' &= f_2(y_1, y_2, \ldots, y_n, x) \\ &\vdots \\ y_n' &= f_n(y_1, y_2, \ldots, y_n, x) \end{aligned} \tag{7.1}$$

As explained in appendix A, a set of n^{th}-order ordinary differential equations may be transformed into a system of first order ordinary differential equations. The first order system can be cast (locally) into the above form, $\mathbf{y}' = \mathbf{f}(\mathbf{y}, t)$. The interface to ordinary differential equations solves for the n functions $y_1(x), ..., y_n(x)$. A system of differential equations is solved by first creating an `GenericOde` structure. This `GenericOde` structure can then be solved to any value of the independent variable.

The generic ode structure is created as:

```
GenericOde ode(int no_of_equations, double start_time,
                double_function derivative, double tolerance)
```

The default solution technique used to solve the system of equations in (7.1) is Adam's method. A fifth order Rungc-Kutta and Gear's methods are also provided. A non-default solution method is used as:

```
GenericOde ode(OCE_choice method, int no_of_eqns, double start_time,
              double_function derivative, double tolerance)
```

The system (7.1) is solved to any value of the independent variable x as:

```
ode.solve(double x, Vector y)
```

A differential equation data structure is destroyed and the memory taken by the data structure is freed by

```
ode.free();
```

Interface to Optimization Problems

A general constrained-optimization problem may be stated as:

$$\begin{array}{lll} \text{minimize:} & F(\mathbf{x}) & \mathbf{x} \in \Re^n \\ \\ \text{subject to:} & l_i \leq x_i \leq u_i & i = 1, 2, \ldots, n \\ & l_i \leq \mathbf{A}\mathbf{x} \leq u_i, & i = n+1, n+2, \ldots, n+n_L \\ & l_i \leq c_i(\mathbf{x}) \leq u_i & i = n+n_L+1, \ldots, n+n_L+n_N \end{array}$$

where

$$\begin{array}{rcl} \mathbf{x} & = & \text{an } n \text{ element vector of variables} \\ F(\mathbf{x}) & = & \text{objective function to be minimized} \\ \mathbf{A} & = & \text{an } n_L \text{ by } n \text{ matrix of constants} \\ c_i(\mathbf{x}) & = & \text{a non linear constraint function} \\ l_i & = & \text{lower bound on the respective variable or constraint function} \\ u_i & = & \text{upper bound on the respective variable or constraint function} \end{array}$$

By selecting one or more of n_L or n_N to be zero, this general constraint optimization problem can be reduced to a non-constrained optimization problem or a constrained-optimization problem with only linear constraints. A number of interfaces for optimization of functions are available. To perform an unconstrained minimization of a function, a user can use:

```
OCE_optimize(int no_of_variables,
             void (*&obj_fun)(), /* routine to compute
                                    objective function and gradient */
             double &optimum_value, /* minimum value of function */
             double x[]             /* variable values at optimum */
            )
```

The function to compute objective function and gradient is declared as

```
void obj_fun(int no_of_variables, double x[], double &func_value,
             double obj_gradient[]);
```

For the general constrained optimization problem with non-linear constraints, the interface provided is as follows.

```
OCE_optimize(int n,                 /* no of variables   */
             int nclin,             /* no of linear constraints */
             int ncnln,             /* no of non-linear constraints */
             double a[],            /* linear constraint matrix */
```

```
double l[/* n+nclin+ncnln */],
double u[/* n+nclin+ncnln */],
                        /* lower and upper bounds */
void (*&obj_fun)(),  /* routine to compute
                          objective function and gradient */
void (*&obj_fun)(),  /* routine to compute
              nonlinear constraints and their jacobian */
double &optimum_value, /* minimum value of function */
double x[]             /* variable values at optimum */
)
```

When a symbolic mathematics interface is fully integrated into OCE, the interface to the solution of optimization problems will become simpler, with gradients and jacobians computed automatically.

7.2.3 Partitioning

The support for partitioning is designed to provide the capability of decomposing a problem into subproblems, solving the subproblems and composing the subsolutions. Partitioning support in OCE is provided for communication between different constraint methods approaches.

Communication between Constraint Methods

Different constraint approaches can operate on different objects in the constraint environment. To enable constraint approaches to cooperate with each other, a general sufficiently implementation of objects is required. In the current version of OCE, a number of useful objects have been implemented. The interface provided to any object is a natural mathematical one, rather than being specific to any constraint technique. Using the class derivation facilities of C++, technique specific classes of a general object can be derived if necessary. This general definition of the object enables good communication between the different constraint approaches.

The definitions for vector, matrix and rotation objects in the generic numerical methods layer already support partitioning, by providing a uniform interface independent of the internal representation of the objects. We have defined additional objects that are used by constraint techniques at "higher" level of representation of the constraint problem. For each of the additional objects, a base class definition is created; specific instances of the object are derived from the base definition to create more specialized objects.

PATHS

A base definition of a path in OCE is expressed in terms of n piecewise parametric functions:

$$\mathbf{r}_i(t) = \mathbf{f}_i(t), \;\; t_{i-1} < t < t_i, \;\; 1 < i < n$$

Given a value of t between t_0 and t_n, a path can return position or derivatives of various orders of $\mathbf{r}(t)$ with respect to t. Classes are provided that reparameterize paths and compute the parameters of the Frenet-Serret apparatus [FAUX and PRATT 79]. The member functions provided for a generic path are:

```
GenericPath p;
Vector3 p.position(double t); /* position at parameter value t */
Vector3 p.tangent(double t);  /* tangent at parameter value t */
Vector3 p.normal(double t);   /* normal at parameter value t */
Vector3 p.binormal(double t); /* binormal at parameter value t */
double  p.length(double t);   /* path length to parameter value t */
double  p.curvature(double t);/* curvature at parameter value t */
```

Some specific types of paths are implemented as subclasses of the `GenericPath` class. A piecewise bezier path and a piecewise linear path are inherited classes of a `GenericPath` class and may be respectively created by:

```
BezPath bpath(int no_of_control_pts, Vector3 control_pts);
LinPath lpath(int no_of_control_pts, Vector3 control_pts);
```

An interface to a curve editor also exists. A curve from the curve editor may be read as:

```
CedPath cpath(char *curve_file);
```

A path may be reparametrized by arclength to create another path as:

```
newpath = path.ReParmArcLength()
```

FORCE

Two types of forces are currently defined, `PointForce` and `FieldForce`. A `PointForce` acts on a point on a body. In addition to a force at the center of mass a `PointForce` may generate a torque if the point of application is not the center of mass. A `FieldForce` is a uniform force field that generates only a force at the center of mass. Gravitational force (assumed uniform) is an example of a `FieldForce`[2].

[2]In general, body forces like gravity, surface forces like viscosity and line forces like surface tension are examples of field forces.

```
FieldForce gravity(Vector3(0.0, 0.0, -9.8))
PointForce f1(Vector3 force_vector, Vector3 appl_point)
    /* apply force_vector to the point appl_point in body
       coordinates */
```

RIGID BODY

A generic rigid body object is defined with the following attributes:

```
Center of mass vector
A rotation
Mass of the body
Inertia tensor in body coordinates
Linear Momentum vector
Angular Momentum vector
```

Specific bodies, such as cylinders and spheres have been derived from the `GenericRigidBody`.

```
RigidCylinder()     /* generate a unit cylinder of unit mass */
RigidCylinder(double mass, double radius, Vector3 p1, Vector3 p2)
                    /* Cylinder from point p1 to p2 */
RigidSphere()       /* Unit sphere of unit mass */
RigidSphere(double mass, double radius, Vector3 center)
```

To compute the motion of the rigid bodies, we need to add forces and torques to the definition of rigid bodies. A rigid body used in a dynamic simulation may be derived from the `GenericRigidBody` as:

```
class DynamicRigidBody : public GenericRigidBody{
    Force ExtForces[];
    Torque ExtTorques[];
}
```

FLEXIBLE BODY

A model of flexible solids has been implemented. Currently, the model employs a simple mass-spring model. Springs are modeled as linear-hookean springs with force proportional to the displacement from original length. Examples of simulations using this model are presented in the next chapter. A flexible body may be created as:

```
FlexibleBody(double mass, Vector3 (*&position_func)(),
             int nu, int nv, int nw, double k);
```

The function `position_func` is defined as:

```
Vector3 position_func(double u, double v, double w)
```

and returns the world coordinates of the parametric solid parameterized by body coordinates `u, v, w`.

The body is discretized into `nu`, `nv`, and `nw` mass points in the three parametric directions. Hexahedrons are created from adjacent points with springs between each pair of points. The spring constant for each spring is `k`.

Constraint Approaches

Using the objects defined above, we have implemented a number of constraint approaches. These approaches can be used in conjunction to generate the simulations demonstrated in the next chapter.

INVERSE DYNAMICS

The inverse dynamics technique involves computing constraint forces that together with external forces acting on bodies will produce motion of bodies according to specified constraints. The implementation of this technique in OCE is based in part on [BARZEL and BARR 88]. The implementation supports both rigid bodies and a simple elasticity model of flexible bodies. Simulations may mix rigid bodies and flexible bodies.

Classes defining forces, paths, rigid bodies and flexible bodies defined in OCE have been used in the definition of the inverse dynamics system. Objects of these classes may be generated or used by other constraint approaches as well as by the inverse dynamics system. In this way, one constraint approach may be plugged with other constraint approaches. For instance, in the next section, we describe, how we can create paths by optimizing functionals of the path. Also, in the next chapter, we present an example in which an optimized path is used in conjunction with inverse dynamics.

The derivation of inverse dynamics as presented in [BARZEL and BARR 88] leads to a linear system of equations for the constraint forces in the system as:

$$\mathcal{M}\mathcal{F}_c + \mathcal{B} = 0$$

where

$$\begin{aligned} \mathcal{M} &= \text{a sparse, possibly non-square matrix} \\ \mathcal{F}_c &= \text{vector of constraint forces} \\ \mathcal{B} &= \text{a vector} \end{aligned}$$

The behavior of the system of rigid bodies is determined by solving the equations of motion of rigid bodies as described in appendix B. The equations form a set of ordinary differential equations.

Using the sparse matrix representations and solutions of differential equations provided in the generic numerical methods layer, we have implemented an inverse dynamics system in OCE.

An inverse dynamics system of objects is created by

```
InverseDynamics system;
```

The rigid and flexible bodies in the inverse dynamics system can be created as:

```
ID_Nail(Vector3 position) /* fixed point in space */
ID_FlexibleBody(double mass, Vector3 (*&position_func)(),
        int nu, int nv, int nw, double k);  /* Flexible body */
ID_Cylinder(double mass, double radius, Vector3 p1, Vector3 p2)
                    /* Cylinder from point p1 to p2 */
ID_Sphere(double mass, double radius, Vector3 center)
ID_Point(Vector3 position, ID_Body body)
```

Flexible bodies and rigid bodies in the inverse dynamics system are derived from a general class, `ID_Body` and therefore, a rigid body or flexible body may be used to create an `ID_Point`.

External forces may be defined as

```
Field_Force gravity(Vector3 force_vector)
Point_force f1(Vector3 position, Vector force_vector)
```

and applied to a body as:

```
body.add_force(Force force)
```

Three types of constraints can currently be declared:

```
system.PointToNail(ID_Nail n, ID_Point p)
system.PointToPoint(ID_Point p1, ID_Point p2)
system.PointToPath(ID_Point n, Path path)
```

The inverse dynamics system may be simulated by:

```
system.simulate(start_time, end_time, time_step)
```

Examples of simulations carried out with the inverse dynamics system in OCE are presented in the next chapter.

7.2.4 Temporal Sequencing

Support for temporal sequencing is provided in OCE through classes for event-units and event-graphs. An event-graph simulator is a member function of the class for an event-graph. An event-unit is created as

```
EventUnit e1(char *name,
             int (*&behavior_rule)(),
             int (*&compute_state_at_event)())
```

Once event-units have been defined, an event-graph can be created by connecting event-units as:

```
EventGraph eg;
eg.connect(EventUnit e1, EventUnit e11)
eg.connect(EventUnit e1, EventUnit e11, EventUnit e12)
    etc.
```

In a `connect` message, the first argument is the current behavior rule that a system of objects is simulating in. The remaining arguments represent respectively the next behavior rule for each of the events that can occur in the current behavior rule.

An event-graph is simulated simply as

```
eg.simulate(start_time, end_time, time_step)
```

7.2.5 Syntactical Extensions

We have added some syntactical extensions to C++ to provide features to aid in programming simulations.

Dimensions of Physical Objects

We have added declaration of the dimension of an object to the object's declaration. The dimension declaration is added as part of the class definition in C++.

DIMENSION DECLARATION SYNTAX

The declaration is introduced by the keyword **dimension** in the class declaration and has the following syntax.

```
<dimension declaration> ::= dimension <dimension declarator>
<dimension declarator> ::= <dimension letter>[<dimension power>]
<dimension letter> ::= M | L | T
<dimension power> ::= [-]<unsigned integer>
```

The declarations are declared as powers of **M**, **L**, and **T** for mass, length, and time respectively. The dimension letter is followed by an optional integer representing the exponent. If the power is missing, a unit exponent is assumed.

For example a class `force` may be declared as:

```
class force: public vector{
dimension: MLT-2;
  /* other data and functions */
};
```

This declaration declares `force` to be a vector with dimensions of MLT^{-2}.

Dimension Usage

When parsing an expression, dimension declarations are used to check dimensional integrity of the equation. For each object in the expression that has dimensions defined, dimensions are checked for compatibility.

The dimension check is done during preprocessing of a program. There is no run time penalty because of dimensional checking. The dimension check also does not cause any syntax change in an expression.

7.2.6 User Interface

This section discusses the user interface provided to OCE. Textual interface is provided as basic input interface using a general programming language. Some special purpose interfaces also exist such as interface to an interactive editor for path specification. Extensive rendering and animation support is also provided.

Textual Environment

The basic interface is through programs written in our extension of C++ as described in section 7.1. Currently, the C++ program is compiled and linked with the OCE environment. A C++ interpreter is under development that will allow the simulation programs to be interpreted. After programs are debugged using the interpreter, they can be migrated into the compiled part of the system for efficiency.

Rendering Support

The visible results of OCE in most cases are computer images or animation. OCE has extensive graphics support to make it easy to incorporate rendering and animations into simulation results. We have attempted to provide a functional rendering subsystem that handles most of the rendering needs of a simulation. According to the basic philosophy of OCE, access is still provided to the basic processes of the rendering subsystem so that a user can implement rendering supports that are not already available.

Each object in OCE can be rendered, since all objects in OCE are derived from a base object called `OCE_Object`, which has a virtual rendering function defined for it. All derived objects can overload this function with their own rendering function. The overloaded render functions may use the facilities provided in the OCE Graphics substrate to generate images and animations.

The rendering system provides support in two ways,

- Creation and maintenance of organizations of renderable objects
- Rendering primitive or organizations of objects

To provide the above support, the following facilities are provided in the rendering subsystem.

Objects, Object Lists, Object Hierarchies

Various canonical geometrical objects are supported in the render subsystem. Some of the objects are polygons, quadric surfaces, cylinders, superquadrics, and parametric surfaces. A renderable object may be created out of basic objects by organizing them in structures such as render-lists and render-hierarchies as described below.

A render-list is a list of renderable objects. Render lists can be passed to the render subsystems to be rendered. A render-hierarchy is hierarchy in the fashion of the common graphics **push-pop hierarchy** [FOLEY and VANDAM 82]. The lists and hierarchy are dynamic ones and they can be modified as needed during simulation.

Surface Descriptions

The render-subsystem provides definition and usage of various surface types. To create a renderable representation of itself, an object can ask for a surface resource from the rendering system and associate it with its geometry.

Making an image or an animation

The rendering system contains a list of renderable objects which might contain primitive objects, render-lists or render-hierarchies. At each frame time, all the objects in the render list are rendered. The renderer works in two modes, online and off-line.

The online interactive renderer is a **z-buffer** based renderer. The objects on the render's list are rendered using hardware that uses the z-buffer rendering algorithm. **Drop shadows** are supported to provide additional depth cues.

The off-line renderer outputs data to a file for off-line, and possibly, high quality renderers such as **ray tracers**. A ray tracer program exists that can input the off-line rendering output of OCE and render high-quality images providing real shadows, reflection and refraction.

To create animations, an interface is provided with an ABEKAS A60 digital frame recorder. When the animation mode is activated, the interface records each frame on the ABEKAS A60.

Camera Object

The renderer has a general camera model that provides viewing control on the world scene. Various parameters of the camera such as the point at which the camera is looking, the position of the camera and the field of view are accessible to other parts of the simulator. In this way the camera can be controlled directly by the results of a simulation. For example, the camera look-at point may be constrained to be always on a particular object so that the object is always centered on the screen. Similarly, the camera position may be constrained to follow user specified trajectories. Trajectories are general paths as described earlier.

7.3 Summary

In this chapter we have discussed the facilities available in OCE and their implementation. Using an approach of making building blocks that plug together, we have been able to construct a prototype system that provides considerable simulation capabilities. A number of built-in objects are provided to provide support of creating mathematical structures, creating physically based simulation structures and rendering outputs. Using the prototype system we have performed some simulations and built some simulation packages. We will present examples from these simulations in the next chapter.

Chapter 8

Examples of Constraints

In the previous chapters, we have presented our approach for designing a unified constraint-based simulation system. In chapter 7, we described a prototype system, OCE, that we have implemented based on our design concepts. In this chapter we present simulations that we have carried out using OCE. Some of these problems have been presented in previous chapters.

8.1 Building A Package on OCE Layers

The first three simulation examples demonstrate the ability to build a constraint package on top of the facilities provided in OCE. We have implemented an inverse dynamics package fashioned after [BARZEL and BARR 88]. The technique in [BARZEL and BARR 88] produces the constrained motion of rigid bodies by computing forces that under Newtonian mechanics laws would move the bodies according to the constraints. We have implemented an extended version which also includes flexible models. Currently, we use a mass-spring model for flexible bodies based on an approximation to the metric tensor of a solid. A flexible body is discretized as point masses connected by hookean springs. The implementation uses a differential equation solver and a sparse linear equation solver. The inverse dynamics system was implemented first for rigid bodies alone. The availability of high level layers to solve constraint problems helped to compose the simulations in a short time.

Two simulation sequences are shown in figure 8.1 and 8.2[1]. The first sequence shows rigid bodies being assembled. Constraints are imposed between points fixed in the coordinate system of rigid bodies and between "nails", points fixed in world space.

The second sequence (figure 8.2) shows constraints between two flexible bodies. One point of one flexible body is attached to one nail and one point on the second

[1]The animation sequences advance left to right and top to bottom.

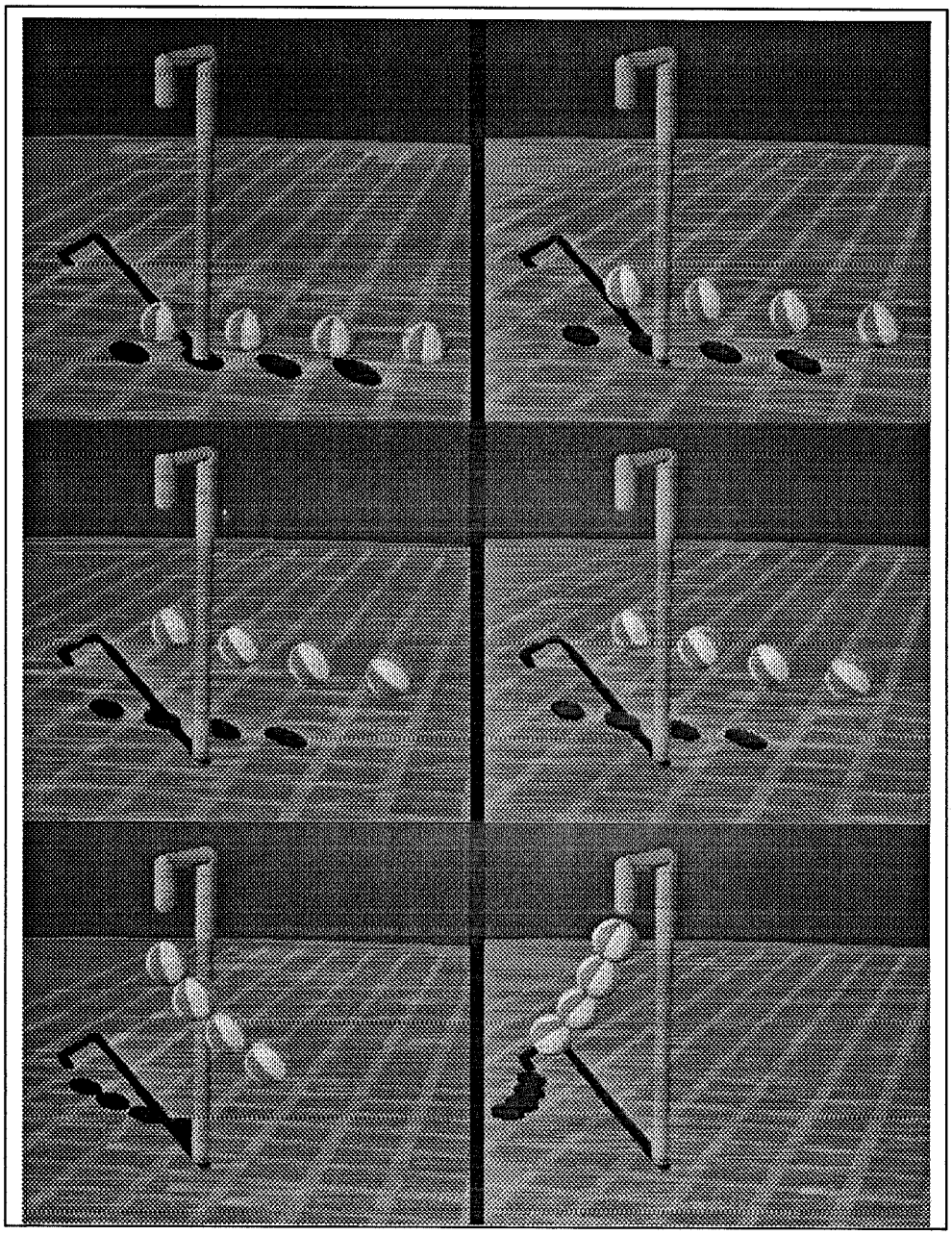

Figure 8.1: A rigid body sequence produced from an inverse-dynamics package built on top of OCE vertical refinement layers.

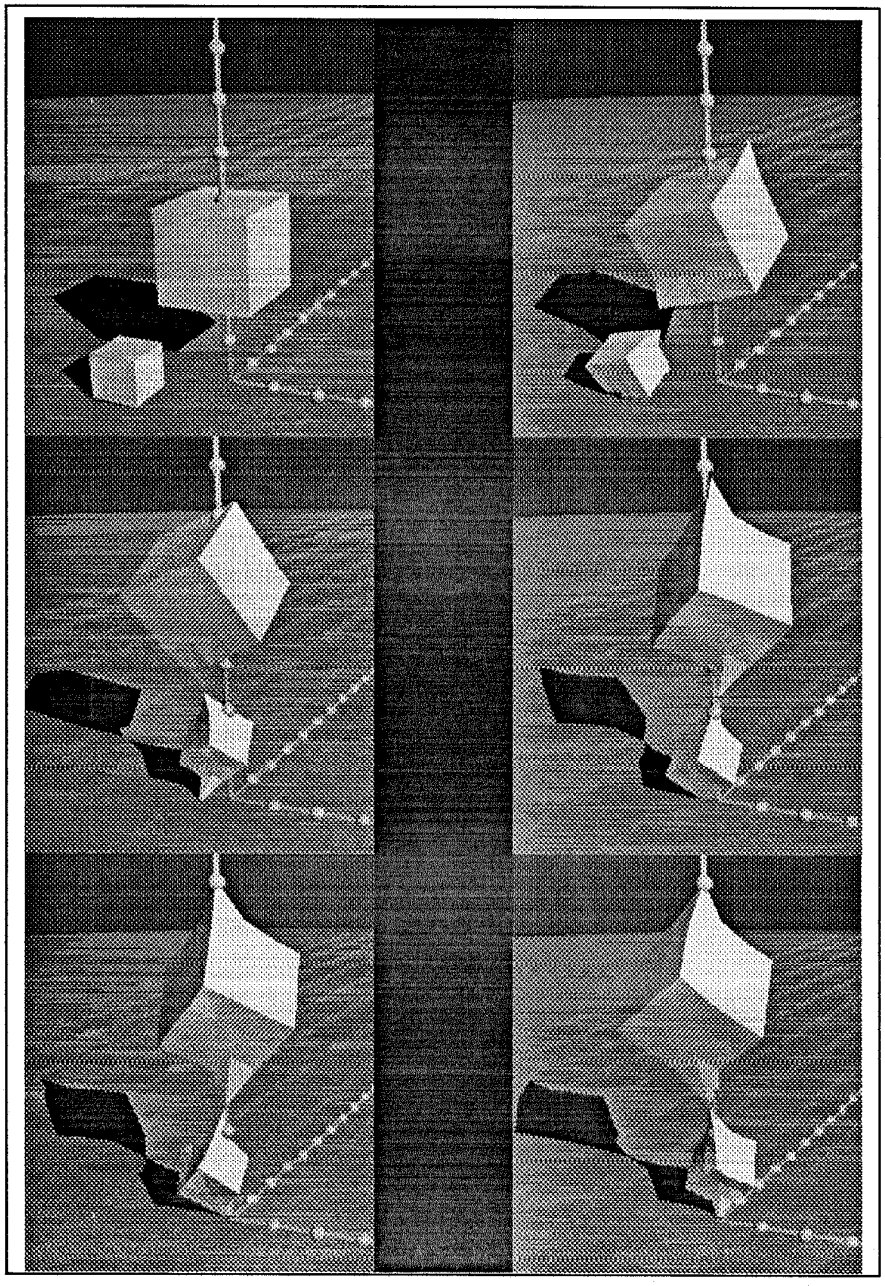

Figure 8.2: A flexible body sequence produced from an inverse-dynamics package built on top of OCE vertical refinement layers.

flexible body is attached to a second nail. Further, a point of the first body is connected to a point on the second body. The bodies start at positions where the constraints are not met. The bodies then move under the forces computed by inverse-dynamics to meet the constraints.

The code segment that generates the animation segment is:

```
  InverseDynamics system;
  ID_Nail n1(Vector3(n1x, n1y, n1z));/* Declare two fixed points */
  ID_Nail n2(Vector3(n2x, n2y, n2z));

/* Declare two flexible cubes by specifying their corner points */

  ID_FBody fb1(mass1, left_bottom_1, right_top_1);
  ID_FBody fb2(mass2, left_bottom_2, right_top_2);
  system.add_body(fb1);
  system.add_body(fb2);

  ID_Point p1(left_bottom, fb1);     /* Declare anchor points on */
  ID_Point p2(right_top, fb1);       /* Flexible Bodies          */
  ID_Point p3(left_bottom, fb2);
  ID_Point p4(right_top, fb2);

  system.PointToNail(n1, p1);        /* Declare Constraints */
  system.PointToNail(n2, p4);
  system.PointToPoint(p2, p3);

  system.simulate(start_time, end_time, t_step);
```

8.1.1 Heterogeneous Objects

The sequence in figure 8.3 shows the interaction between rigid and flexible bodies. A flexible body is constrained to connect to the end point of a rigid pendulum which in turn is connected to a nail. Inverse dynamics with rigid and flexible bodies is used to compute forces that move the flexible body and the rigid body according to the imposed constraints.

The sequence was generated by the following code:

```
  InverseDynamics system;
  ID_Nail nail(Vector3(n1x, n1y, n1z));
  Field_Force gravity(Vector3(0.0, 0.0, -9.8));
```

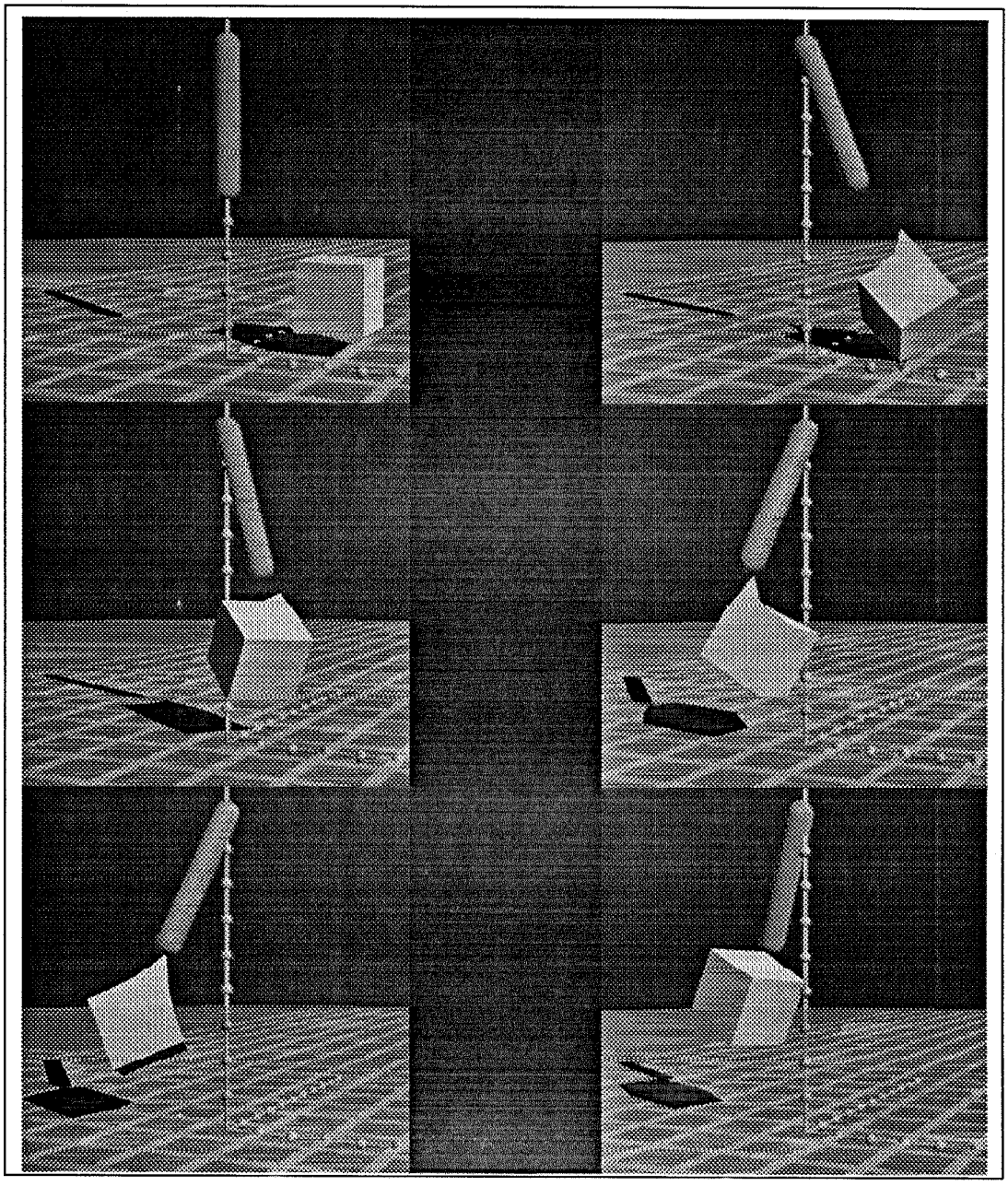

Figure 8.3: A sequence showing the interaction of rigid and flexible objects using inverse dynamics.

```
ID_Cylinder cyl(mass1, radius1, p1, p2);
ID_FBody fb(mass2, left_bottom_2, right_top_2);

cyl.add_force(gravity);
fb.add_force(gravity);

system.add_body(cyl);
system.add_body(fb);

ID_Point cp1(origin, cyl);     /* Declare connecting points   */
ID_Point cp2(end, cyl);        /* on the bodies in the system */
ID_Point fp1(right_top, fb2);

system.PointToNail(nail, cp1);/* Declare Constraints */
system.PointToPoint(cp2, fp1);

system.simulate(start_time, end_time, t_step);
```

8.2 Multiple Solution Methods

The animation sequence described here shows the use of multiple techniques in solving a problem (figure 8.4). The problem is to move an object from point A to point B around obstacles. A three-axis robot is used to move the object. We wish to find the robot joint angles as a function of time that will move the object from point A to point B satisfying the constraints.

Using the terminology in chapter 5 on horizontal partitioning, we break the problem into three sequenced subsystems. That is, we solve the problem as three subproblems one after the other. Each step uses the solution generated by the previous step. This strategy can work only if the representations for objects used by different subsystems are compatible. As described in the previous chapter, we have achieved this compatibility by defining generic classes for the objects. Each technique can use specialized classes derived from these generic classes.

The three subsystems used to solve the problem at hand are:

Subproblem 1. Find a path avoiding the obstacles

Subproblem 2. Move the object on the path

Subproblem 3. Find the joint angles of the robot that will cause the robot to move the object on the computed path

The code generating the simulation in figure 8.5 is as follows:

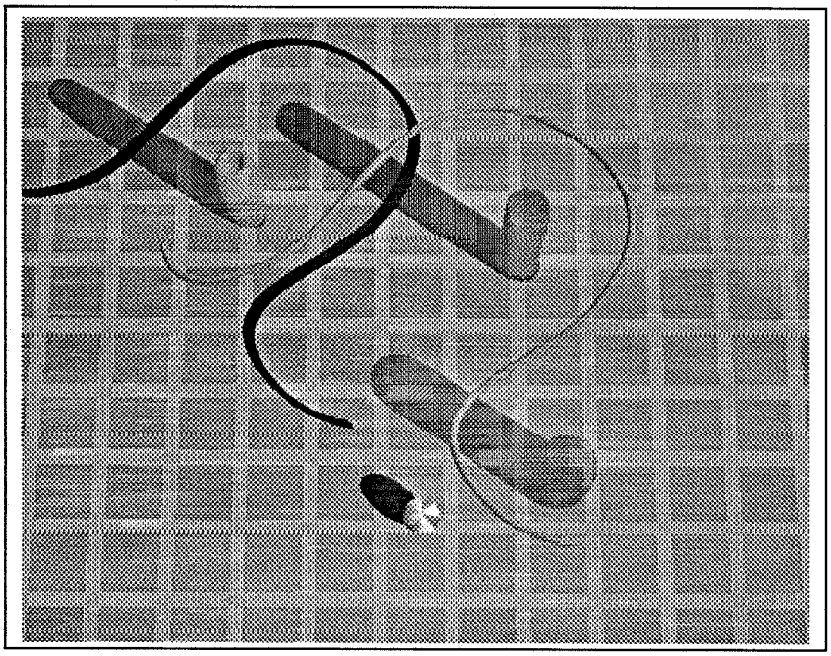

Figure 8.4: A path determined by an optimization procedure to avoid obstacles.

```
  InverseDynamics system;

  IK_robot3 robot;    /* Instantiate a robot */
  Field_Force gravity(Vector3(0.0, 0.0, -9.8));

/* Step 1. Determine path */

  CedPath path1("initpath.ced");    /* Use a curve editor path as
                                       an initial guess */
  OptPath path2(path1, obstacle1, obstacle2, obstacle3);
      /* User written class to optimize length of path1 avoiding
                three obstacles. Path path2 is the result.  */

/* Step 2. Move body on path */

  ID_Sphere sp(b_mass, b_radius, initial_pos);
  sp.add_force(gravity);

  system.add_body(sp);
```

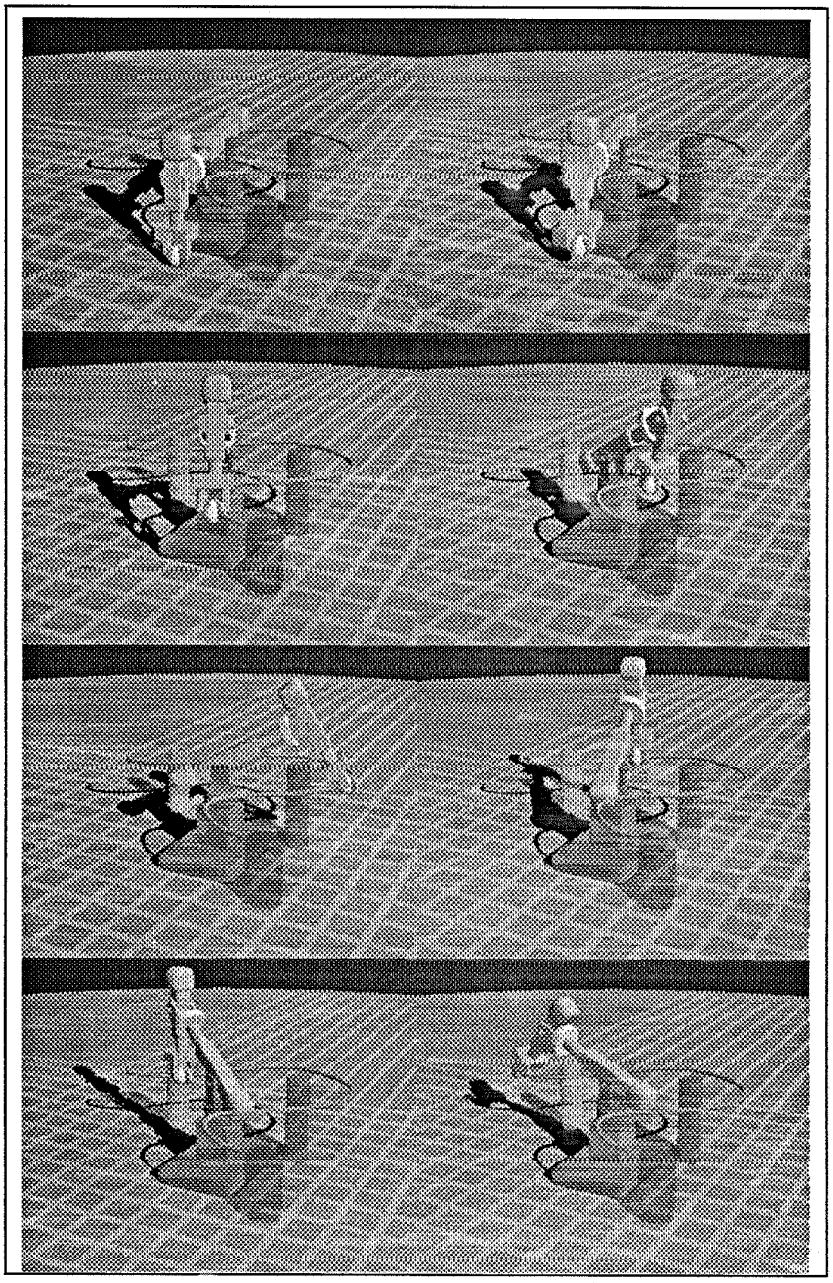

Figure 8.5: An object is moved using inverse dynamics on a path. Inverse kinematics computes joint angles for a robot that move the object along the trajectory computed by inverse dynamics.

```
  ID_Point p2(Vector3(0.0, 0.0, 0.0), sp);
  system.PointToPath(p2, path2); /* Constrain center of ball to the
                                    optimized path */

  for(t=start_time; t<end_time; t+=time_step){
    system.solve(t, t+time_step, time_step);/*Solve position of ball*/
/* Step 3. Compute robot joint angles */
    robot.compute_angles(sp.cm());    /*Compute joint angles for */
                                      /*robot to position ball   */
    system.render();
  }
```

First, the path is determined by optimizing an energy function that minimizes the length of the path while avoiding the obstacles (Figure 8.4). A path generated by a curve editor provided an initial guess for the optimization procedure.

After determining the path, the sphere is moved on the path using inverse-dynamics. We impose a path-to-point constraint between the path determined in step 1, and the center of mass of the sphere.

The robot joint angles are determined by inverse kinematics to follow the ball. A three-axis robot class `IK_robot3` exists in OCE which takes a point in 3-space as input and determines the joint angles.

8.3 Time-Event Simulation

This example illustrates a time-event simulation in a robot work cell (Figure 8.6). The work cell consists of two conveyer belts, C_{in} bringing work-pieces in and C_{out}, taking work-pieces out. A robot picks up pieces from C_{in}, works on the work-piece and deposits it on the out going belt C_{out}. The simulation proceeds as follows:

1. The vision system V on the incoming belt stops the belt when it sees a piece at the end of the belt

2. The robot picks up the piece, carries it to a work bench W.

3. After "working" on the piece on the work bench, the robot picks up the finished piece and puts it on the outgoing belt.

The simulation in (figure 8.7) was created using the time-event approach of chapter 6. Behavior rules for the robot-workcell system between events were created and connected to create a time-graph. The code that generated the sequence in figure 8.7 is as follows:

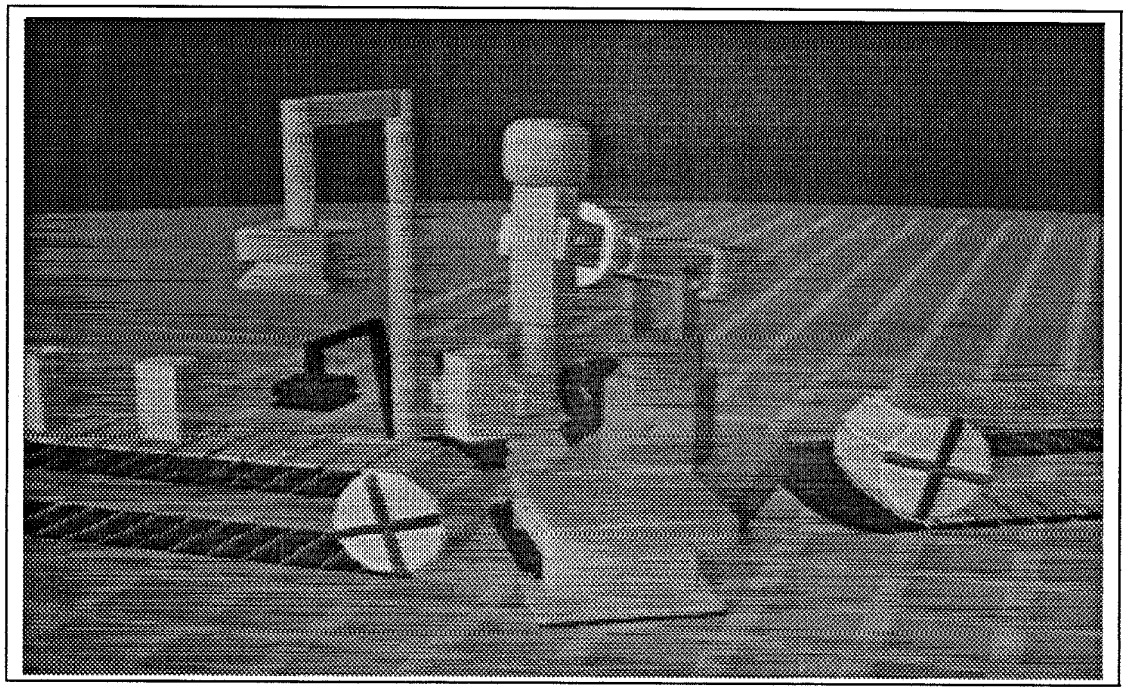

Figure 8.6: A time-event example. Multiple systems in a robot work cell are simulated. Work pieces come in on conveyer belt C_{in}. The pieces are picked up by a robot, worked on and delivered to an outgoing belt C_{out}.

```
/* Create event-units necessary for the simulation */

  EventUnit wait_for_belt1("e1",
                           r_wait_for_belt1_to_stop,
                           compute_state_at_b_stop);
  EventUnit move_to_belt1("e2",
                           r_move_to_belt1,
                           compute_state_at_belt1);
          :                :
          :                :
  EventUnit move_to_home("en",
                           r_move_to_home,
                           compute_state_at_home);

/* Create event graph */
  EventGraph robot_graph;
  robot_graph.connect(wait_for_belt1, move_to_belt1);
  robot_graph.connect(move_to_belt1, grab_object);
```

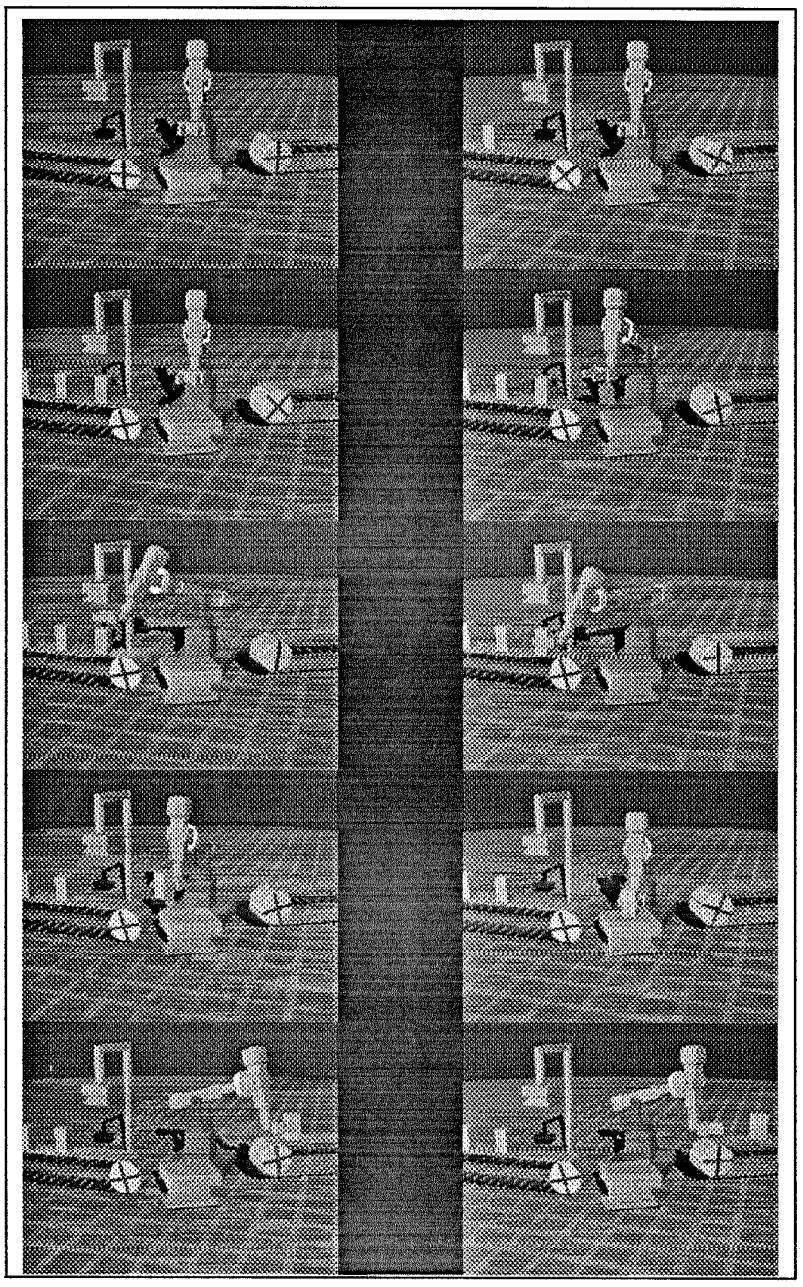

Figure 8.7: Frames from a robot workcell simulation designed as a time-event sequence.

```
          :            :
          :            :
  robot_graph.connect(move_to_belt2, move_to_home);
  robot_graph.connect(move_to_home, wait_for_belt1);

/* Simulate event graph */
  robot_graph.simulate(start_time, end_time, time_step);
```

8.4 Summary

The examples presented in this chapter show some of the capabilities of a prototype system built on the concepts presented in this thesis. The basic philosophy of the design has been identifying various ways in which solutions of subproblems can be plugged together to form overall solutions to a problem. Using partitioning, refinement and temporal sequencing, we have been able to construct a system that provides us this pluggability.

Part IV

Conclusions and Appendices

Chapter 9

Conclusions

Constraint-based modeling techniques are emerging as a useful computer graphics approach for modeling and designing objects and their behaviors. To solve a constraint problem effectively, we often need to break complex problems into subproblems, to use multiple and disparate techniques on the subproblems, and then combine the solutions to subproblems to create the overall solution. In this thesis we have presented an approach to unify constraint-based modeling (figure 1.1).

In chapter 1, we identified a collection of constraint approaches that would be found useful in a constraint environment. The constraint approaches are:

- Inverse kinematics
- Inverse dynamics
- Constrained optimization
- Calculus of Variations
- Simulated annealing
- Hamiltonian (and Lagrangian) physics
- Differential-Algebraic Equations

We then explored the answer to the question, "how do we unify different constraint approaches?" We want the ability to both use the above constraint approaches and to incorporate new constraint approaches in the same framework. We started by defining primitive elements of a constraint environment, objects, constraints and simulation entities.

We then identified different ways in which the primitive elements can interact during the solution of constraint problems. To design a general and extensible constraint environment, we used three general problem solution strategies:

1. Refinement, to step-wise refine a representation of a constraint problem into the most primitive level, basic numerical solution techniques,

2. Partitioning, to decompose a problem into subproblems at the same level of representation, and

3. Temporal Sequencing, to create the behavior of a system of objects during a time interval by organizing the system's behaviors during sub-intervals.

Based on these general strategies, we designed a prototype system and a programming interface for constraint-modeling. Although the prototype system, OCE, implements a subset of the presented design, we can still do fairly complex simulations as described in chapter 8. Guided by partitioning, we have implemented very general representations of objects in our system. These general objects act as interfaces between different techniques enabling disparate techniques to work together. Guided by refinement, we have build a layered structure of techniques that provides different representations for a problem. At higher levels in the layered refinement structure, we can use built-in objects and techniques to build simulations quickly. At the same time, we can built other custom simulations by using lower level facilities in the vertical refinement layers. Temporal sequencing has provided us a model to design complex discontinuous behaviors of systems as sub-behaviors. Since the building blocks in the modeling environment are not dependent or based on a particular technique, the design is easily extensible. New techniques can be added into the framework and can interact with both existing and new techniques.

Some aspects of horizontal partitioning and vertical refinement suggest hardware speedups. Horizontal partitioning and temporal sequencing provide us a problem decomposition model that can use parallel computers to solve parts of problems concurrently. Vertical refinement layers provides us a new simulation pipeline, some parts of which may be migrated into hardware providing fast workstations for constraint-based modeling.

Appendix A

Mathematical Techniques for Simulation

In this appendix we discuss some mathematical techniques that are found useful in formulating and solving simulation problems.

A.1 Cartesian Coordinate Frame Transformations

In formulating physical simulation problems, it is sometimes useful to specify different parts of the problem in different coordinate frames. For example, it might be convenient to specify a point on a rigid body with respect to a coordinate frame fixed to the body with the origin of the frame coinciding with the center of mass of the body. To be able to relate different parts of the problem specified in different coordinate frames, it is then necessary to transform various physical quantities from one frame to another.

In this section we present some transformation results between three-dimensional cartesian coordinate frames.

Consider two cartesian frames A and B moving arbitrarily with respect to each other. Let the position of a particle P measured with respect to frame B be given by ${}^{B}\mathbf{r}(t)$.

Then the motion of P may be expressed in frame A as[1]

$$ {}^{A}\mathbf{r}_{P} = {}^{A}\mathbf{r}_{B_{org}} + {}^{B}\mathbf{r}_{P} \tag{A.1} $$

Frame B may be expressed in terms of frame A as a displacement of the origin of B with respect to the origin of frame A and a rotation. If we denote unit vectors along the x, y, and z axes of a coordinate frame as $\mathbf{e}_1$, $\mathbf{e}_2$, and $\mathbf{e}_3$, then we can define

[1]Using notation similar to [CRAIG 89]

the orientation of frame B with respect to frame A with a rotation matrix as

$$ {}^A_B\mathbf{R} = \left[\begin{array}{ccc} {}^A_B\mathbf{e}_1 & {}^A_B\mathbf{e}_2 & {}^A_B\mathbf{e}_3 \end{array} \right] = \left[\begin{array}{ccc} {}^A_Be_{11} & {}^A_Be_{21} & {}^A_Be_{31} \\ {}^A_Be_{12} & {}^A_Be_{22} & {}^A_Be_{32} \\ {}^A_Be_{13} & {}^A_Be_{23} & {}^A_Be_{33} \end{array} \right] \tag{A.2} $$

In terms of the rotation matrix, a position vector in frame B may be transformed to frame A as

$$ {}^A\mathbf{r} = {}^A_B\mathbf{R}\ {}^B\mathbf{r} + {}^A\mathbf{r}_{B_{org}} $$

The velocity of point P in frame A is given by

$$ {}^A\dot{\mathbf{r}} = {}^A\mathbf{v}_P \ = \ {}^A\mathbf{v}_{B_{org}} + {}^A_B\mathbf{R}\ {}^B\mathbf{v}_P + {}^A_B\omega \times ({}^A_B\mathbf{R}\ {}^B\mathbf{r}) \tag{A.3} $$

The acceleration of point P in frame A is given by

$$ \begin{aligned} {}^A\dot{\mathbf{v}}_P = {}^A\mathbf{a}_P \ &= \ {}^B\mathbf{a}_{B_{org}} + {}^A_B\mathbf{R}\ {}^B\mathbf{a}_P + 2\,{}^A_B\omega \times {}^A_B\mathbf{R}\ {}^B\mathbf{v}_P + {}^A_B\dot{\omega} \times {}^A_B\mathbf{R}\ {}^B\mathbf{r} \\ &\quad + {}^A_B\omega \times ({}^A_B\omega \times {}^A_B\mathbf{R}\ {}^B\mathbf{r}) \end{aligned} \tag{A.4} $$

Point fixed in a Rigid Body

If frame B is fixed with respect to a rigid body and point P is fixed in frame B, there is a significant simplification in the expressions above. For such a fixed point P

$$ {}^B\mathbf{v}_P = {}^B\mathbf{a}_P = 0 $$

and the expressions for velocity and accelerations for a point in frame A are

$$ {}^A\mathbf{v}_P \ = \ {}^A\mathbf{v}_{B_{org}} + {}^A_B\omega \times ({}^A_B\mathbf{R}\ {}^B\mathbf{r}) \tag{A.5} $$

$$ {}^A\mathbf{v}_P \ = \ {}^B\dot{\mathbf{v}}_{B_{org}} + {}^A_B\dot{\omega} \times {}^A_B\mathbf{R}\ {}^B\mathbf{r} + {}^A_B\omega \times ({}^A_B\omega \times {}^A_B\mathbf{R}\ {}^B\mathbf{r}) \tag{A.6} $$

Assuming that all vectors are specified with respect to frame A, the velocity and acceleration expressions may be further simplified to

$$ \begin{aligned} \mathbf{v} \ &= \ \mathbf{v}_{B_{org}} + \omega \times \mathbf{r} \\ \mathbf{a} \ &= \ \mathbf{a}_{B_{org}} + \dot{\omega} \times \mathbf{r} + \omega \times \omega \times \mathbf{r} \end{aligned} $$

A.2 Quaternions

A quaternion is a mathematical structure that may be used to represent orientations in a compact form.

A.2.1 Definition

A quaternion is a four component structure that may be thought of as having a scalar part and a three component vector part. So a quaternion may be represented as:

$$\mathbf{q} = [s, \mathbf{v}] \tag{A.7}$$

where

$$\begin{aligned} s &= \textit{a scalar} \\ \mathbf{v} &= \textit{a three component vector} \end{aligned}$$

A quaternion may also be represented as a four component structure much like a complex number may be represented as a two component structure. In this notation,

$$\mathbf{q} = w + x\mathbf{i} + y\mathbf{j} + z\mathbf{k}$$

and

$$\begin{gathered} \mathbf{ii} = \mathbf{jj} = \mathbf{kk} = -1 \\ \mathbf{ij} = \mathbf{k} \quad \mathbf{jk} = \mathbf{i} \quad \mathbf{ki} = \mathbf{j} \\ \mathbf{ji} = -\mathbf{k} \quad \mathbf{kj} = -\mathbf{i} \quad \mathbf{ik} = -\mathbf{j} \end{gathered}$$

A.2.2 Quaternion Algebra

Two quaternions may be added and multiplied as:

$$\begin{aligned} \mathbf{q}_1 &= [s_1, \mathbf{v}_1] \\ \mathbf{q}_2 &= [s_2, \mathbf{v}_2] \\ \mathbf{q}_1 + \mathbf{q}_2 &= [s_1 + s_2, \mathbf{v}_1 + \mathbf{v}_2] && \text{(A.8)} \\ \mathbf{q}_1\mathbf{q}_2 &= [s_1 s_2 - \mathbf{v}_1 \cdot \mathbf{v}_2, s_1\mathbf{v}_2 + s_2\mathbf{v}_1 + \mathbf{v}_1 \times \mathbf{v}_2] && \text{(A.9)} \end{aligned}$$

Note that both the addition and multiplication formulas can be verified by the four component representation of a quaternion presented above.

The magnitude of a quaternion $q = [s, \mathbf{v}]$ is given by

$$|\mathbf{q}|^2 = s^2 + \mathbf{v}.\mathbf{v} \tag{A.10}$$

The inverse of a quaternion $q = [s, \mathbf{v}]$ is

$$\mathbf{q}^{-1} = \frac{[s, -\mathbf{v}]}{s^2 + |\mathbf{v}|^2} = \frac{[s, -\mathbf{v}]}{|\mathbf{q}|^2} \tag{A.11}$$

A vector may be multiplied by a quaternion by considering the vector to be a quaternion with a zero scalar part. Hence a quaternion $\mathbf{q} = [s, \mathbf{v}]$ is multiplied with a vector $\mathbf{p}$ as

$$\mathbf{qp} = [-\mathbf{v} \cdot \mathbf{p}, s\mathbf{p} + \mathbf{v} \times \mathbf{p}]$$

A.2.3 Quaternion as a Rotation

A quaternion can be used to represent a rotation by an angle θ about an axis $\mathbf{v}$. The vector $\mathbf{v}$ should be a unit vector. Such a rotation is given by:

$$q = [\cos(\theta/2), \mathbf{v}\sin(\theta/2)]$$

A vector $\mathbf{p}$ is rotated by a quaternion to the vector $\mathbf{p}'$ as:

$$\mathbf{p}' = \mathbf{q}\mathbf{p}\mathbf{q}^{-1}$$

A.2.4 Converting a Quaternion to a Rotation Matrix

Using the four component representation of a unit magnitude quaternion as

$$\mathbf{q} = w + x\mathbf{i} + y\mathbf{j} + z\mathbf{k}, \textit{where } w^2 + x^2 + y^2 + z^2 = 1$$

the corresponding rotation matrix is given by

$$\mathbf{R} = \begin{bmatrix} 1-2y^2-2z^2 & 2xy-2wz & 2zx+2wy \\ 2xy+2wz & 1-2z^2-2x^2 & 2yz-2wx \\ 2zx-2wy & 2yz+2wx & 1-2x^2-2y^2 \end{bmatrix}$$

A.3 Dual of a vector

The dual of a vector $\mathbf{b} = [b_1, b_2, b_3]^T$ is the antisymmetric matrix $\mathbf{b}^*$:

$$\mathbf{b}^* = \begin{bmatrix} 0 & -b_3 & b_2 \\ b_3 & 0 & -b_1 \\ -b_2 & b_1 & 0 \end{bmatrix}$$

With the above definition, for a vector $\mathbf{a}$,

$$\begin{aligned} \mathbf{b}^*\mathbf{a} &\equiv \mathbf{b} \times \mathbf{a} \\ \mathbf{b}^{*T}\mathbf{a} &\equiv \mathbf{a} \times \mathbf{b} \\ \mathbf{b}^{*T} &\equiv -\mathbf{b}^* \\ \mathbf{b}^*\mathbf{b} &\equiv \mathbf{0} \end{aligned}$$

A.4 Numerical Solution of Ordinary Differential Equations

Problems involving ordinary differential equations (ODEs) can be reduced to problems involving sets of first order differential equations by a variable substitution. Therefore,

the generic problem in ODEs can be reduced to the study of a set of n coupled first order differential equations, locally of the form:

$$\begin{aligned} y_1' &= f_1(y_1, y_2, \ldots, y_n, x) \\ y_2' &= f_2(y_1, y_2, \ldots, y_n, x) \\ &\vdots \\ y_n' &= f_n(y_1, y_2, \ldots, y_n, x) \end{aligned}$$

To fully specify a problem involving differential equations, we also need boundary conditions. Boundary conditions may be broadly divided into two categories:

Initial Value Problems The value of all the y_i is given at some starting value x_s and we wish to solve for all $y_i(x)$ for $x > x_s$.

Boundary Value Problems The values of y_i is given at more than one value of x. We wish to solve for $y_i(x)$ such that the specified values of y_i at the specified x's agree.

We will describe the solution of initial value problem here.

The simplest solution method for an initial value problem is obtained by using the first order approximation of a derivative. This method is called the Euler method. The formula for the Euler method is:

$$\mathbf{y}^{n+1} = \mathbf{y}^n + h\mathbf{f}(x^n, \mathbf{y}), \quad n = 0, 1, 2, \ldots$$

The superscript i denotes the values at the end of the i-th step. The formula advances the solution through an interval h using the derivative information at the beginning of the interval. The Euler method is $O(h^2)$. The Euler method is not very accurate as compared to other fancier methods for the same step size and may not be stable for stiff systems of equations.

The Runge-Kutta method uses multiple evaluations of the derivative functions over an interval to approximate a Taylor series expansion to a higher order. For example, the fourth order runge-kutta method ($O(h^5)$) uses four evaluations of the derivative and proceeds as:

$$\begin{aligned} \mathbf{k}_1 &= h\mathbf{f}(x^n, \mathbf{y}^n) \\ \mathbf{k}_2 &= h\mathbf{f}(x^n + \frac{h}{2}, \mathbf{y}^n + \frac{\mathbf{k}_1}{2}) \\ \mathbf{k}_3 &= h\mathbf{f}(x^n + \frac{h}{2}, \mathbf{y}^n + \frac{\mathbf{k}_2}{2}) \\ \mathbf{k}_4 &= h\mathbf{f}(x^n + h, \mathbf{y}^n + \mathbf{k}_3) \\ \mathbf{y}^{n+1} &= \mathbf{y}^n + \frac{\mathbf{k}_1}{6} + \frac{\mathbf{k}_2}{3} + \frac{\mathbf{k}_3}{3} + \frac{\mathbf{k}_4}{6} + O(h^5) \end{aligned}$$

[PRESS et al 88] also contains the description and code for an adaptive-step size method for runge-kutta.

Adam's Method to solve differential equations belongs to the category of *predictor-corrector* methods. Predictor-corrector methods record past function values and extrapolate them, using polynomial extrapolation, to predict what the next step would yield. Using this predicted value of the dependent variable, a corrected value is computed by the corrector step.

For example, the Adams-Bashford-Moulton method computes the predictor step as:

$$\mathbf{y}^{n+1} = \mathbf{y}^n + \frac{h}{24}(55\mathbf{f}^n - 59\mathbf{f}^{n-1} + 37\mathbf{f}^{n-2} - 9\mathbf{f}^{n-3}) + O(h^5)$$

The predicted value of $\mathbf{y}^{n+1}$ is used to compute a corrected value by:

$$\mathbf{y}^{n+1} = \mathbf{y}^n + \frac{h}{24}(9\mathbf{f}^{n+1} + 19\mathbf{f}^n - 5\mathbf{f}^{n-1} + \mathbf{f}^{n-2}) + O(h^5)$$

Note that to compute $\mathbf{f}^{n+1}$, the value of $\mathbf{y}^{n+1}$ is required.

[GEAR 71] and [PRESS et al 88] present more details of solution methods for differential equations.

A.5 Calculus of Variations

Calculus of variations involves the determination of a curve such that a given line integral along the curve has a stationary value.

Given a function

$$y = y(x), \tag{A.12}$$

and the functional

$$J[y] = \int_{x_0}^{x_1} F(x, y, y')dx \tag{A.13}$$

J[y] is minimized if

$$-[F]_y = \frac{d}{dx}F_{y'} - F_y = 0 \tag{A.14}$$

$$y''F_{y'y'} + y'F_{y'y} + F_{y'x} - F_y = 0 \tag{A.15}$$

In general,

$$\delta J = \int_{x_0}^{x_1} [F]_y \delta y dx + Fy'\, \delta y\Big|_{x_0}^{x_1} \tag{A.16}$$

$$\frac{\delta J[y]}{\delta y} = \frac{\partial F}{\partial y} - \frac{d}{dx}\left(\frac{\partial F}{\partial y'}\right) \tag{A.17}$$

More generally, if

$$J[y] = \int_{x_0}^{x_1} F(x, y, y', \ldots, y^{(N)}) dx \tag{A.18}$$

then,

$$\frac{\delta J[y]}{\delta y} = \sum_{n=0}^{N} (-1)^n \frac{d^n}{dx^n} \left(\frac{\partial F}{\partial y^{(n)}} \right) \tag{A.19}$$

where $y^{(n)} = \frac{d^n y}{dx^n}$.

If

$$J = \int_{\Omega} F(\mathbf{x}, y, y_{x_1}, y_{x_2}, \ldots) d\mathbf{x} \tag{A.20}$$

where the integral is now a multiple integral over $\mathbf{x} \in \Omega$ and $y_{x_i} = \frac{\partial y}{\partial x_i}$.

In this case, the variational derivative is

$$\frac{\delta J[y]}{\delta y} = F_y - \sum_i \frac{\partial}{\partial x_i} F_{y_{x_i}} \tag{A.21}$$

On the other hand if

$$J[y] = \int_{x_0}^{x_1} F(x, y, z, \ldots, y', z', \ldots) dx \tag{A.22}$$

$$\frac{\delta J}{\delta y} = \frac{\partial F}{\partial y} - \frac{d}{dx} \left(\frac{\partial F}{\partial y'} \right) \tag{A.23}$$

$$\frac{\delta J}{\delta z} = \frac{\partial F}{\partial z} - \frac{d}{dx} \left(\frac{\partial F}{\partial z'} \right) \tag{A.24}$$

Appendix B

Constraint Satisfaction Techniques

In this appendix, we present some techniques that might are useful in devising constraint satisfaction strategies for physically-based modeling. (The techniques of calculus of variations and techniques for solution of differential equations have already been discussed in appendix A.)

B.1 Rigid Body Dynamics

In classical mechanics, the motion of rigid body under the application of forces and torques is governed by the so called *Newton-Euler equations of rigid body motion.* These equations relate inertial properties of a body to the applied forces and torques. The state variables associated with a rigid body are:

- **Mass** m: The mass of a body is a scalar quantity.
- **Rotational Inertia Tensor I**: The rotational inertia tensor is a 3×3 matrix that relates a body's angular velocity ω to the angular momentum of the body. The inertia tensor depends on the coordinate frame that it is expressed in. Representing the three axes of a cartesian coordinate frame as x_1, x_2 and x_3, the various components of the inertia tensor are given by:

$$\begin{aligned} I_{11} &= \int_V (x_2^2 + x_3^2)\rho \, dv \\ I_{22} &= \int_V (x_3^2 + x_1^2)\rho \, dv \\ I_{33} &= \int_V (x_1^2 + x_2^2)\rho \, dv \\ I_{12} = I_{21} &= \int_V (x_1 x_2)\rho \, dv \\ I_{23} = I_{32} &= \int_V (x_2 x_3)\rho \, dv \end{aligned}$$

$$I_{31} = I_{13} = \int_V (x_3 x_1)\rho \, dv$$

where ρ is the mass density of the body. It is usually convenient to compute the inertia tensor $\mathbf{I}$ with respect to a coordinate frame fixed with respect to the body. Given the body inertia, $\mathbf{I}_{body}$, the transformation of the inertia tensor to another coordinate frame is given by

$$\begin{aligned} \mathbf{I}^{-1} &= \mathbf{R}\,\mathbf{I}_{body}^{-1}\,\mathbf{R}^T \\ \mathbf{I} &= \mathbf{R}\,\mathbf{I}_{body}\,\mathbf{R}^T \end{aligned}$$

where $\mathbf{R}$ represents the rotation matrix of the body coordinate system with respect to world coordinates.

- Center of Mass of the Body, $\mathbf{x}_{CM}$: The center of mass is a fixed vector in the body coordinate system of the rigid body given by:

$$\mathbf{x}_{CM} = \frac{1}{m}\int_V \mathbf{r}\, d\rho$$

- Orientation of the body: The orientation of the body may be represented by a rotation matrix $\mathbf{R}$ or a quaternion $\mathbf{q}$ as discussed in appendix A. The use of quaternions is usually preferable from the point of view of stability of the solution of the differential equations of motion.
- Linear Momentum $\mathbf{p}$
- Angular Momentum $\mathbf{L}$
- Linear Velocity $\mathbf{v}$
- Angular Velocity ω
- Net Force $\mathbf{F}$: The net force is the sum of all the forces $\mathbf{F}_i$ applied to the body.

$$\mathbf{F} = \sum_i \mathbf{F}_i$$

- Net Torque $\mathbf{T}$: The net torque is the sum of all the torques $\mathbf{T}_i$ and the sum of torques due to all the forces $\mathbf{F}_i$ applied to the body.

$$\mathbf{T} = \sum_i \mathbf{r} \times \mathbf{F}_i + \sum_j \mathbf{T}_j$$

where $\mathbf{r}$ is the position vector of the point of application of the force with respect to the center of mass of the body.

With the above definitions and symbols, the newton-euler equations of motion governing the motion of a rigid body are:

$$\frac{d\mathbf{x}_{CM}}{dt} = \mathbf{v} = \frac{1}{m}\mathbf{p} \tag{B.1}$$

$$\frac{d\mathbf{p}}{dt} = \mathbf{F} \tag{B.2}$$

$$\frac{d\mathbf{q}}{dt} = \frac{1}{2}\omega\mathbf{q} = \frac{1}{2}\mathbf{I}^{-1}\mathbf{L}\mathbf{q} \tag{B.3}$$

$$\frac{d\mathbf{L}}{dt} = \mathbf{T} \tag{B.4}$$

B.2 Inverse Kinematics

Kinematics is the study of motion of bodies without consideration of the physical causes of the motion.

Let the state of a system of bodies be represented as $\mathcal{X}$ and let the geometric motion of a body described as

$$P = f(\mathcal{X}).$$

Given a specific instance of a desired motion, P_d, the **inverse kinematics** problem involves determination of instance(s) of the state $\mathcal{X}$ which would generate the desired motion.

Solution of inverse kinematics problems, generally, involves solving algebraic equations. [CRAIG 89] discusses inverse kinematics in the context of robotics.

B.3 Inverse Dynamics

Forward dynamics involves the determination of motion of bodies under the influence of applied forces and torques for rigid, flexible, or fluid objects. The motion of rigid bodies is determined by using Newton's equations of rigid body motion that relate linear and angular accelerations of bodies to the forces and torques applied through inertial properties of the bodies, namely, mass and the inertia tensor [GOLDSTEIN 80]. The equations of rigid body motion are second order ordinary differential equations. Similarly, equations from elasticity theory can be used to determine motion of flexible bodies and Navier-Stokes equation may be used to model fluids.

Inverse Dynamics is the inverse problem of Forward Dynamics. Inverse dynamic techniques compute the necessary forces and torques on a body that will result in a desired motion or equilibrium state. The forces and torques are computed by using the specification of the motion and the forward dynamics laws that apply to the bodies.

Problems in inverse dynamics generally involve solving differential-algebraic equations. Examples of inverse dynamics are presented in chapters 4 and 8. [BARZEL and

BARR 88] employs teh techniques of inverse dynamics to create a rigid body modeling system.

B.4 Constrained optimization

A optimization problem involves minimizing a function (an *objective function*) of several variables, possibly subject to restrictions on the values of the variables defined by a set of *constraint functions.*

Optimization problems may be classified into particular categories depending on the properties of the objective and constraint functions. Some of the categories for an objective function are: non-linear, sum of squares of nonlinear functions, quadratic, sums of squares of linear functions and linear. Similarly some categories of constraints are nonlinear, sparse linear, linear, and bounded (inequality constraints).

A general constraint optimization problem may be stated as:

$$\begin{array}{lll} \text{minimize:} & F(\mathbf{x}) & \mathbf{x} \in \Re^n \\ \\ \text{subject to:} & l_i \leq x_i \leq u_i & i = 1, 2, \ldots, n \\ & l_i \leq \mathbf{Ax} \leq u_i, & i = n+1, n+2, \ldots, n+n_L \\ & l_i \leq c_i(\mathbf{x}) \leq u_i & i = n+n_L+1, \ldots, n+n_L+n_N \end{array}$$

A good discussion of techniques for optimization is presented in [GILL et al 81].

B.5 Simulated annealing

Simulated annealing is a combinatorial method to minimize an objective function. Simulated annealing is useful when the objective function is defined over a discrete configuration space. There is an analogy between simulated annealing and thermodynamics, the way that liquids crystallize or metals cool and anneal. At high temperatures, atoms in a metal move freely with respect to each other. If the liquid is cooled slowly, the atoms pack closely and form a low energy crystal. If the metal is cooled quickly or "quenched," the atoms cannot reach the low-energy crystalline state and form a high energy amorphous state.

Simulated annealing attempts to determine a low energy configuration of a system with a discrete configuration space. The system starts at some "temperature" T. A new configuration C_2 of the system different from the current configuration C_1 is generated. If the energy of the configuration C_1 is E_1 and the energy of configuration C_2 is E_2, the system has a probability

$$p = \exp\left[\frac{-(E_2 - E_1)}{kT}\right]$$

of going to configuration C_2. If $E_2 < E_1$, p is set to unity and the new configuration is always accepted. However, if $E_2 > E_1$, the new configuration is accepted with probability p. The "temperature" T is gradually lowered. In the earlier stages of simulated annealing, the system can jump out of deep energy wells. As a result, the system may be able to jump out of local minima. The scheme of always taking the downward step but occasionaly taking an uphill step is generally called *Metropolis algorithm.* The algorithm involves the definition of:

1. A description of possible system configurations
2. A selector of random configurations from the system configuration space
3. A objective function E of system configuration
4. A control parameter T and an *annealing schedule* which determines how T is lowered

[PAPADIMIRIOU and STEIGLITZ 82] and [KIRKPATRICK, GELATT and VECCHI 83] discuss simulated annealing.

B.6 Lagrangian physics

Lagrangian physics may be considered as the energy analog of newtonian physics. In lagrangian physics, the equations of motion of a system of bodies is determined by differentiating the expressions for energy of the system in terms of the degrees of freedom of the system.

The equations of motion of a system with n degrees of freedom, with q_i representing the i-th generalized variable is given by:

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{q}_j}\right) - \frac{\partial L}{\partial q_j} = 0$$

where the *Lagrangian* L is defined as

$$L = T - V.$$

In the above equation, T represents the kinetic energy of the system and V represents the potential energy of the system expressed in terms of the generalized variables q_i.

[GOLDSTEIN 80] describes lagrangian physics and applications.

Appendix C

Implementation Examples from OCE

In this appendix, we describe the implementation of two general classes in OCE to provide an idea about how simulation entities may be created. The first class described is an inverse dynamics system and the second class is paths in OCE.

The most generic object at the topmost level in OCE is `OCE_Object`, which has virtual functions for rendering, saving and reading the state of an object. Most other objects are derived from `OCE_Object`.

C.1 Inverse dynamics object

The inverse dynamics object, `InverseDynamics` is also derived from `OCE_Object`. The state of the object `InverseDynamics` contains a list of bodies and a list of constraints. The member functions for the object `InverseDynamics` are provided to

1. Create an `InverseDynamics` object
2. Destroy an `InverseDynamics` object
3. Add a body to `InverseDynamics`
4. Add a constraint to `InverseDynamics`
5. Simulate the `InverseDynamics` object

The `InverseDynamics` object is defined as:

```
class InverseDynamics:public OCE_Object{
    gslist(ID_Body) body_list;  // gslist(o_type) is a macro to create
                                // a list of objects of type o_type
    gslist(ID_Constraint) constraint_list;
public:
    InverseDynamics();          // Constructor: Function to create
                                // InverseDynamics object
```

```
    InverseDynamics();          // Destructor: Function to destroy
                                // InverseDynamics object

    void add_body(ID_Body &body);
    void add_constraint(ID_Constraint &constraint);

    void solve_MFB();           // solve for constraint forces using
                                // a sparse matrix solver

    void InverseDynamics::simulate(double start_t, double end_t);
                                // simulate the system by integrating
                                // the equations of motion of the bodies
                                // in the system

    void  render(double t);     // render the objects in the system
};
```

The InverseDynamics object uses objects ID_Body and ID_Constraint. ID_Body is a generic class with ID_RigidBody and ID_FlexibleBody derived from ID_Body. ID_RigidBody also derives from the class GenericRigidBody.

The class definition of GenericRigidBody is as follows:

```
class GenericRigidBody : public OCE_Object {
protected:
    double i_mass;
    Quaternion i_q;             // Rotation
    Matrix33 i_inertia;         // Inertia tensor
    Frame ref_frame;            // Reference Frame
    Vector3 i_cm,               // Center of Mass
            i_velocity,
            i_omega,            // Angular Velocity
            i_p, i_L;           // p=Linear momentum L=angular momentum
public:
    RigidObj(double mass, Matrix33 Inertia); // Constructor

    double mass() return i_mass;            // accessor functions
    Matrix33 inertia() return i_inertia;
    Vector3 velocity()return i_velocity;
        :       :    :
    Vector3 omega()return i_omega;
    Vector3 cm()return i_cm;
```

```
    void set_velocity(Vector3 &vel);          // Modifier functions
        :       :     :
    void set_cm(Vector3 &pos);
    void set_omega(Vector3 &om){} ;
```

The class definition of ID_Body, a body used in InverseDynamics is:

```
class ID_Body {    // Inverse Dynamics Body
protected:
    int i_body_type;
public:
    ID_Body();     // Constructor

    virtual void compute_external_forces(double t);
    virtual void add_force(Force *force){};
    virtual void add_torque(Torque *torque){};
    virtual Vector3 net_force()return(Vector3(0.0,0.0,0.0));
    virtual Vector3 net_torque()return(Vector3(0.0,0.0,0.0));
};
```

A rigid body used in InverseDynamics is derived from both ID_Body and GenericRigidBody as:

```
class ID_RigidBody : public ID_Body, public GenericRigidBody {
    int no_of_constraints;

    int no_of_ext_forces;                  // External forces and
    Force i_ext_force[MAX_FORCES];         // Torques
    int no_of_ext_torques;
    Torque i_ext_torque[MAX_TORQUES];
public:
    ID_RBody(double mass, Matrix33 Inertia);  // Constructor
    void compute_external_forces(double t);
    void add_force(Force *force);
    void add_torque(Torque *torque);
    Vector3 net_force();
    Vector3 net_torque();
};
```

A flexible body ID_FlexibleBody, used in InverseDynamics is derived from ID_Body as:

```
class ID_FlexibleBody : public ID_Body {
```

```
    int i_nu, i_nv, i_nw;   // No of subdivisions in the parametric
                            // directions
    double mass;
    F_Point *m_points;      // State of mass points
    F_Spring *m_springs;    // State of springs between points
    int no_of_constraints;
    int no_of_ext_forces;
    Force i_ext_force[MAX_FORCES];
    int no_of_ext_torques;
    Torque i_ext_torque[MAX_TORQUES];
public:
    ID_FBody(double lmass, Vector3 &c1, Vector3 &c2, int nx, int ny,
                  int nz, double k);    // Constructor
    void render(double t);
    void compute_external_forces(double t);
    void add_force(Force *force);
    void add_torque(Torque *torque);
    Vector3 net_force();
    Vector3 net_torque();
};
```

A flexible body `ID_FlexibleBody` is simulated as point masses interconnected with springs. Point masses are declared as the class `F_Point` and springs as `F_Spring`.

```
class F_Point : public ID_Body {   // Point mass in flexible
                                   // body
    int no_of_constraints;
    Vector3 i_position, i_momentum;
    double i_mass;
    F_Spring *i_springs;   // Springs attached to mass
public:
    Vector3 net_force();
    Vector3 net_torque();
    Vector3 momentum();     // Accesor Functions
    Vector3 velocity();
    Vector3 position();
    void set_momentum(Vector3 m);  // Modifier Functions
    void set_position(Vector3 p);
};

class F_Spring{       // Spring in flexible body
```

```
    double i_k,      // Spring Constant
           i_l0;     // Spring rest length
    F_Point *m1, *m2; // Masses connected to spring
public:
    double spring_const();
    double spring_length();
    Vector3 spring_force();
};
```

Constraints in InverseDynamics are derived from a generic constraint class, ID_Constraint. A constraint needs to compute forces that are caused by the deviation of the constraint from its satisfied state.

```
class ID_Constraint : public SObject{
protected:
    int l_no_of_bodies;   // no of bodies effected by constraint
    ID_Body *l_bodies[MAX_BODIES]; // Bodies effected by constraint
public:
    virtual int compute_force(double t){};
        // Compute force components due to constraint
};
```

A point to nail constraint stipulates that a point on a body should be connected to a nail, a point fixed in space. A point to point constraint stipulates connection between two points on two bodies or the same body, and point to path constraint constrains a point on a body to follow a path. The classes for these three types of constraints are:

```
class PointToNail:public ID_Constraint{
    Nail *nail;        // In World Coordinates
    ID_Point *point;   // In body coordinates
    double Tau;        // Time constant of constraint
public:
      PointToNail(InverseDynamics &system, Nail &n, ID_Point &p);
        // Constructor
      int compute_force_components(double t);
};

class PointToPath:public ID_Constraint{
    Path *i_path;
    ID_Point *i_point; // In body coordinates
    double Tau;
public:
    PointToPath(InverseDynamics &system, ID_Point &p,
```

```
        GenericPath &path);
    int compute_force_components(double t);
};

class PointToPoint:public ID_Constraint{
    ID_Point *point1, *point2;  // In body coordinates
    double Tau;
public:
    PointToPoint(InverseDynamics &system, ID_Point &p1, ID_Point &p2);
    int compute_force_components(double t);
};
```

C.2 Path object

A path is frequently used object in simulations. A generic path object is defined in OCE. Specialized paths are derived from the generic path.

The generic path has virtual member functions for position, tangent, normal, binormal, length and curvature.

```
class GenericPath : public SObject{
protected:
  double start_t, end_t;
public:
    virtual Vector3 position(double t);
    virtual Vector3 tangent(double t);
    virtual Vector3 normal(double t);
    virtual Vector3 binormal(double t);
    virtual double  length(double t);
    virtual double  curvature(double t);
    GenericPath ReParmArcLength();  // Reparametrize by arclength
    void render(double t);
};
```

A piecewise bezier path is derived from `GenericPath` by overloading the member functions and declaring data specific to a bezier path.

```
class BezierPath : public GenericPath{
protected:
    int no_of_control_points;
    int no_of_curves;
    Vector3 *control_point;
    Vector4 *xcoeffs;
    Vector4 *ycoeffs;
    Vector4 *zcoeffs;
```

```
    double *t;
public:
    BezPath(int no, Vector3 *points, double *times);
    virtual Vector3 position(double t);
    virtual Vector3 tangent(double t);
    virtual Vector3 normal(double t);
    virtual Vector3 binormal(double t);
    virtual double  length(double t);
    virtual double  curvature(double t);
};
```

Other paths like `LinearPath` (piecewise linear) and `CedPath` are derived from `GenericPath` in a similar way. Note that a path, irrespective of type has the same interface.

References

[ARMSTRONG and GREEN 85] *Dynamics for Animation of Characters with Deformable Surfaces,* William W. Armstrong and Mark W. Green, Visual Computer, 1985.

[BARAFF 89] *Analytical Methods for Dynamic Simulation of Non-penetrating Rigid Bodies,* David Baraff, Computer Graphics, Vol. 23, No.3, July 1989.

[BARR 88] **Introduction to Physically-based Modeling**, Alan H. Barr, Course Chairman, Siggraph Course Notes 1988.

[BARR 84] *Geometric Modeling and Fluid Dynamic Analysis of Swimming Spermatozoa,* Alan H. Barr, PhD Thesis, Department of Mathematical Sciences, RPI, Troy, NY.

[BARR and BARZEL 90] *Models of Discontinuous Phenomena,* Alan H Barr and Ronen Barzel, To appear.

[BARTELS, BEATTY and BARSKY 83] *An Introduction to the use of Splines in Computer Graphics,* Richard H. Bartels, John C. Beatty and Brian A. Barsky, Tech Report No. UCB/CSD 83/136, Computer Science Division, University of California Berkeley, 1983.

[BARZEL 91] *Modeling Heterogeneous Objects,* Ronen Barzel, PhD Thesis, Caltech, To appear.

[BARZEL and BARR 90] *Structured Modeling,* Ronen Barzel and Alan H Barr, To appear.

[BARZEL and BARR 88] *A Modeling System Based on Dynamic Constraints,* Ronen Barzel and Alan H Barr, Computer Graphics, Vol. 22, No. 4, August 1988, pp. 179-188.

[BORNING 79] *Thinglab – A constraint-oriented simulation laboratory,* Alan Borning, Report SSL-79-3, Xerox Palo Alto Research Center, Palo Alto, CA. July 1979.

[BORNING 81] *The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory,* Alan Borning, ACM Transactions on Programming Languages and Systems, Vol. 3, No. 4, October 1981, Pp 353-387.

[BORNING 86] *Defining Constraints Graphically,* Alan Borning, CHI '86 Proceedings, April 1986, pp 137-143.

[BOYCE and DEPRIMA 77] **Elementary Differential Equations and Boundary Value Problems**, William E. Boyce and Richard C. DiPrima, John Wiley & Sons, New York, 1977.

[BRATLEY, FOX and SCHRAGE 83] **A Guide to Simulation**, Paul Bratley, Bennett L. Fox and Linus E. Schrage, Springer-Verlag, New York.

[COHEN & GREENBERG 86] *The Hemi-Cube: A Radiosity For Complex Environments,* M.F. Cohen and D.P. Greenberg, Computer Graphics, July 1985, pp. 31-40.

[CRAIG 89] **Introduction to Robotics, Second Edition,** John J. Craig, Addison-Wesley Publishing Company, Reading, Mass. (1989).

[DENAVIT and HARTENBERG 55] *A Kinematic Notation for Lower-Pair Mechanisms Based on Matrices,* J. Denavit and R. S. Hartenberg, Journal of Applied Mechanics, June 1955, pp. 215-221.

[DEO 74] **Graph Theory with Applications to Engineering and Computer Science**, Narsingh Deo, Prentice-Hall International Inc., Englewood Cliffs, N.J.

[FAUX and PRATT 79] **Computational Geometry for Design and Manufacture**, I. D. Faux and M. J. Pratt, John Wiley & Sons, New York.

[FOLEY and VANDAM 82] **Fundamentals of Interactive Computer Graphics**, J. D. Foley and A. Van Dam, Addison Wesley Publishing Company 82.

[GEAR 71] **Numerical Initial Value Problems in Ordinary Differential Equations**, William C. Gear, Prentice-Hall, Englewood Cliffs, NJ, 1971.

[GOLDBERG and ROBSON 83] **Smalltalk-80: The Language and its Implementation**, Adele Goldberg and David Robson, Addison-Wesley Publishing Company, Inc., Reading, Mass. (1983).

[GOLDSTEIN 80] **Classical Mechanics**, *second edition,* Herbert Goldstein, Addison-Wesley Publishing Company, Inc., Reading, Mass. (1980).

[GILL et al 81] **Practical Optimization**, P. E. Gill, W. Murray, M. H. Wright, Academic Press, London (1981).

[GRUVER and SOROKA 88] *Programming, High Level Languages,* W. Gruver and B. Soroka, **International Encylopedia of Robotics**, R. Dorf and S. Nof, Editors, Wiley Interscience, 1988.

[IMMEL, COHEN & GREENBERG 87] *A Radiosity Method for Non-Diffuse Environments,* D.S. Immel, M.F. Cohen and D.P. Greenberg, Computer Graphics, August 1986, pp. 133-142.

[ISAACS & COHEN 87] *Controlling Dynamic Simulation with Kinematic Constraints, Behavior Functions and Inverse Dynamics,* Paul M. Isaacs and Michael F. Cohen, Computer Graphics, Vol. 21, No. 4, July 1987.

[HAHN 88] *Realistic Animation of Rigid Bodies,* J. K. Hahn, Computer Graphics, Vol. 22, No. 4, August 1988, pp. 299-308.

[KAJIYA 86] *The Rendering Equation,* James T Kajiya, Computer Graphics, August 86, pp. 143-150.

[KAJIYA and KAY 89] *Rendering Fur with Three Dimensional Textures,* James T Kajiya and Timothy L Kay, Computer Graphics, July 89, pp. 271-280.

[KALRA 90a] *Time and Events in Computer Animation,* Devendra Kalra, To appear.

[KALRA 90b] *A Constraint-Based Figure-Maker,* Devendra Kalra, To appear in the proceedings of Eurographics 90.

[KERNIGHAN and RITCHIE 78] **The C Programming Language,** Brian W. Kernighan and Dennis M. Ritchie, Prentice Hall, Englewood Cliffs, NJ.

[KIRKPATRICK, GELATT and VECCHI 83] *Optimization by Simulated Annealing,* S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, Science, Vol 220, pp. 671-680.

[de KLEER and SUSSMAN 78] *Propagation of Constraints Applied To Circuit Synthesis,* Johan de Kleer and Gerald Jay Sussman, Memo No. 485, AI laboratory, MIT, September 1978.

[KNUTH 84] **The TEX book**, Donald E. Knuth, Addison-Wesley Publishing Company, 1984.

[LELER 87] **Constraint Programming Languages**, Wm Leler, Addison-Wesley Publishing Company, 1987.

[LINTON et al 89] *Composing User Interfaces with InterViews,* Mark A. Linton, John M. Vlissides, and Paul R. Calder, Computer, February 1989.

[LIPPMAN 89] **C++ Primer,** Stanley B. Lippman, Addison-Wesley Publishing Company, 1989.

[MACSYMA 77] **MACSYMA Reference Manual,** The Mathlab Group, MIT Laboratary For Computer Science, Cambridge, Massachusetts, 1977.

[MATHEMATICA 88] **Mathematica, A system for Doing Mathematics by Computer,** Stephen Wolfram, Addison-Wesley Publishing Company, 1988.

[MEAD 89] **Analog VLSI and Neural Systems,** Carver Mead, Addison-Wesley Publishing Company, Reading, MA.

[MILLER 88] *The Motion Dynamics of Snakes and Worms,* Gavin S. P. Miller, Computer Graphics, Vol. 22, No. 4, August 1988.

[MOORE and WILHELMS 88] *Collision Detection and Response for Computer Animation,* Matthew Moore and Jane Wilhelms, Computer Graphics, Vol. 22, No. 4, August 1988.

[MUJTABA and GOLDMAN 81] *AL Users' Manual,* Shahid Mujtaba and Ron Goldman, Report No. STAN-CS-81-889, Department of Computer Science, Stanford University, Stanford, CA 94305.

[NAG 89] *NAG Fortran Library,* The Numerical Algorithms Group Limited, Mayfield House, Oxford, UK OX2 7DE.

[NELSON 85] *Juno, a constraint-based graphics system,* Greg Nelson, Computer Graphics, Vol. 19, No. 3, July 1985, pp. 235-243.

[NYE 89] *Xlib Programming Manual,* Adrian Nye, O'Reilly & Associates, Inc., 1989.

[O'DONNEL and OLSON 81] *GRAMPS- A Graphics Language Interpreter for Real-Time, Interactive, Three-Dimensional Picture Editing and Animation,* T. J. O'Donnel and A. J. Olson, Computer Graphics, Vol. 15, July 1981.

[PAPADIMIRIOU and STEIGLITZ 82] **Combinatorial Optimization: Algorithms and Complexity,** C. H. Papadimiriou and K. Steiglitz, Prentice-Hall, Englewood Cliffs, NJ.

[PENTLAND and WILLIAMS 89] *Good Vibrations: Modal Dynamics for Graphics and Animation,* Alex Pentland and John Williams, Computer Graphics, Vol. 23, No. 3, July 1989, pp 215-222.

[PLATT 89] *Constraint Methods for Neural Networks and Computer Graphics,* John Platt, PhD dissertation, Caltech-CS-TR-89-07, Department of Computer Science, California Institute of Technology.

[PLATT and BARR 88] *Constraint Methods for Flexible Models,* John C. Platt and Alan H. Barr, Computer Graphics, Vol. 22, No. 4, August 1988, pp. 279-288.

[PRESS et al 88] **Numerical Recipes in C**, William H. Press, Brian P Flannery, Saul A Teukolsky and William T Vettering, Cambridge University Press.

[RAIBERT 86] *Legged Robots,* Marc H. Raibert, CACM 29,6(1986) pp 499-514.

[REYNOLDS 87] *Flocks, Herds and Schools: A Distributed Behavioral Model,* Craig Reynolds, Computer Graphics, Vol. 21, No. 4, July 1987.

[REYNOLDS 82] *Computer Animation with Scripts and Actors,* Craig Reynolds, Computer Graphics, Vol. 16, No. 3, July 1982.

[SIMS and ZELTZER 87] *A Figure Editor and Gait Controller for Task Level Animation,* Karl Sims and David Zeltzer, Computer Graphics and Animation Group, August 1987.

[SHAMES 82] **Engineering Mechanics: Statics and Dynamics**, Irving H. Shames, Prentice Hall, Englewood Cliffs, NJ.

[SHIMANO, GESCHKE and SPALDING 84] *Val II: A Robot Programming Language and Control System,* B. Shimano, C. Geschke and C. Spalding, SME Robots VIII Conference, Detroit, June 1984.

[SHOEMAKE 85] *Animating Rotation with Quaternion Curves,* Ken Shoemake, Computer Graphics, July 1985, pp. 245-254.

[STROUSTRUP 85] **The C++ Programming Language,** Bjarne Stroustrup, Addison-Wesley Publishing Company, Reading, Mass. (1985).

[STERN 83] *Bboop–A System for 3D Keyframe Figure Animation,* Garland Stern, Tutorial Notes: Introduction to Computer Animation, Siggraph, July 1983.

[STURMAN 84] *Interactive Keyframe Animation of 3-D Articulated Motion,* David Sturman, Proc Graphics Interface '84, May 1984.

[SUSSMAN and STEELE 80] *CONSTRAINTS–A Language for Expressing Almost-Hierarchical Descriptions,* Gerald Jay Sussman and guy Lewis Steele Jr., Artificial Intelligence, 14(1980), Pp. 1-39.

[SUTHERLAND 63] *Sketchpad, A man-machine graphical communication system,* Ivan E. Sutherland, PhD dissertation, Department of Electrical Engineering, M.I.T., Cambridge, Mass., 1963.

[TAI and MILLER 89] *IC-processed Electrostatic Synchronous Micromotors,* Yu-Chong Tai and Richard S. Muller, Sensors and Actuators, Vol 20, 1989, pp 49-55.

[TAYLOR, SUMMERS and MEYER 82] *AML: A Manufacturing Language,* R. H. Taylor, P. D. Summers and J. M. Meyer, The International Journal of Robotics Research, Vol. 1, No. 3, Fall 1982.

[TERZOUPOULOS and FLEISCHER 88] *Modeling Inelastic Deformation: Viscoelasticity, Plasticity, Fracture,* Demetri Terzoupoulos and Kurt Fleischer, Computer Graphics, Vol. 22, No. 4, August 1988.

[TERZOUPOULOS et al 87] *Elastically Deformable Models,* Demetri Terzoupoulos, John Platt, Alan Barr and Kurt Fleischer, Computer Graphics, Vol. 21, No. 4, July 1987.

[WILHELMS 87] *Using Dynamic Analysis for Realistic Animation of Articulated Bodies,* Jane Wilhelms and Brian Barsky, Graphics Interface, 1985.

[WITKIN, FLEISCHER & BARR 87] *Energy Constraints on Parametrized Models,* Andrew Witkin, Kurt Fleischer and Alan Barr, Computer Graphics, Vol. 21, No. 4, July 1987.

[WITKIN and KASS 88] *Spacetime Constraints,* Andrew Witkin and Michael Kass, Computer Graphics, Vol. 22, No. 4, August 1988.

[ZELTZER 84] *Representation and Control of Three Dimensional Computer Animated Figures,* David Zeltzer, PhD Thesis, The Ohio State University, 1984.

[XTOOLKIT 88] *Programming With The HP X Widgets,* Computer Manual, Hewlett Packard Company, Oregon.

Glossary

Adam's Method: A predictor-corrector method for the solution of ordinary differential equations. This method uses values of the derivative at old values of the independent variable to predict a new value of the dependent variable by extrapolation. A corrected value of the dependent variable is computed using the predicted value.

Animation: The process of creating an illusion of motion by displaying a sequence of images in sequence.

Assembly Language: A symbolic notation for the lowest level operations in a computer usually providing for direct manipulation of processor registers and memory.

Bicubic Patch: A parametric surface defined by cubic equations of two parameters, u and v. The curves generated by varying only one parameter u or v, is a cubic parametric curve of that parameter. The general form of the equation is:

$$\begin{aligned}\underline{P}(u,v) = \; & \underline{A_{11}}u^3v^3 + \underline{A_{12}}u^3v^2 + \underline{A_{13}}u^3v + \underline{A_{14}}u^3 + \\ & \underline{A_{21}}u^2v^3 + \underline{A_{22}}u^2v^2 + \underline{A_{23}}u^2v + \underline{A_{24}}u^2 + \\ & \underline{A_{31}}uv^3 + \underline{A_{32}}uv^2 + \underline{A_{33}}uv + \underline{A_{34}}u + \\ & \underline{A_{41}}v^3 + \underline{A_{42}}v^2 + \underline{A_{43}}v + \underline{A_{44}}\end{aligned}$$

where each of the underlined quantities is a 3×1 vector, each row representing one of the dimensions x, y, z.

Bit-mapped displays: A computer graphics display where the display region is made up of individually addressable dots, called pixels.

Bounding Box: A rectangular box used to enclose another object.

Bounding Volume: A generalization of a *bounding box* where an arbitrary closed region may be used to enclose another object.

Computer animation: The process of creating animation using computers.

Conjugate Gradient: A numerical technique to minimize a scalar function.

Constrained Optimization: The process minimizing or maximizing an objective function subject to constraints.

Constraint-based Modeling: Modeling the behavior of objects using constraints among objects and goals that the objects must reach.

Coriolis Force: A force generated due to the motion of a body in a non-inertial coordinate frame.

Deformation: A modeling operation that alters the shape of an object. Some deformations are bending, twisting, tapering, scaling.

Deviation Function: A function that generates a measure of the deviation of a constrained system from a state in which the constraints are met.

Divide and conquer: A solution strategy in which a problem is broken down into smaller problems which are much more efficient to solve than a big problem. Quicksort, a sorting algorithm is an example.

Drop Shadows: A shadow created by projecting the outline of an object to a shadow plane from the view point of a point light source.

Dynamics: The study of motion of bodies due to physics such as forces and torques.

Energy Function: A scalar function of the state of a system. Most often, energy functions are defined such that the desired state of the system occurs at the minimum value of the energy function.

Euler Angles: A scheme to represent the rotation of a body through three successive rotations.

Finite Differences: The technique to approximate a function by linear functions between discretized samples of the function.

Hessian matrix of a function: The Hessian matrix of a function

$$f(\mathbf{x}) = f(x_1, ..., x_n)$$

is the $n \times n$ matrix

$$H_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}$$

Gear's Method: A numerical solution technique for stiff differential equations.

Graphics pipeline: A sequence of transformation steps that transform a world space description of a polygon into a two-dimensional image on a discrete pixel screen.

Inverse Dynamics: The process of computing forces which when applied to bodies would lead to a desired motion.

Jacobian matrix: The Jacobian matrix of a vector of functions

$$\mathbf{f}(\mathbf{x}) = [f_1(x_1, ..., x_n), ..., f_m(x_1, ..., x_n)]^T$$

is the $m \times n$ matrix

$$J_{ij} = \frac{\partial f_i}{\partial x_j}$$

In other words, the i^{th} row of the matrix $\mathbf{J}$ is the gradient of $f_i(\mathbf{x})$.

Key-frames: Important frames in an animation. Intermediate frames are created by interpolating the state of objects from key frames.

Kinematic Modeling: "Physicsless" modeling performed through mathematical structures such as functions or numbers.

Kinematics: The process of studying the motion of objects without considering the physical causes motion.

Minimal least squares solution: The solution of a non-square system $\mathbf{Ax} = \mathbf{b}$ that minimizes the value of $|\mathbf{Ax} - \mathbf{b}|^2$.

Lagrange method: A method to derive equations of motion of objects from the expressions of energy of the objects.

LU-Decomposition: A numerical technique to compute the solution of linear systems of equations and to invert a matrix.

Modeling: Modeling refers to the creation of mathematical models that represent collections of objects.

NTSC: An acronym for *National Television Standards Committee.* The term NTSC is now used for a television broadcast standard that was formulated by this committee. The standard involves transmitting 30 frames per second. Each frame is composed of two fields each composed of 262.5 lines forming a total of 525 lines for the whole frame.

Parametric Surface: A surface defined in terms of an explicit function of some parameters as:

$$x = f_x(u_1, ..., u_n)$$

$$y = f_y(u_1, ..., u_n)$$

$$z = f_z(u_1, ..., u_n)$$

Physically-Based Modeling: A modeling technique in which objects and their behavior is modeled by modeling the physics of the objects.

Pixel: An acronym for picture element, a pixel represents a the smallest addressable unit on the screen of a graphics display. A computer graphics picture is composed of a rectangular array of pixels.

Ray Tracing: A rendering scheme in which the color of any *pixel* in the image is computed by tracing a light ray through the pixel. A ray is shot into the world and intersected with the objects in the world. Each intersection spawns more rays that are used to simulate for example, reflection, and refraction. A shadow ray casting determines if a particular light source illuminates the point of intersection. At each intersection, the net intensity is the composite of the intensities of the rays spawned at that intersection and the intersections of these secondary rays.

This technique enable various phenomena to be simulated such as reflection, refraction, shadows and with certain extensions penumbrae, depth of field, and diffuse shadows.

Rendering: The process of converting mathematical descriptions of objects and scenes into a two-dimensional image. Rendering simulates the interaction of light rays with the objects forming the scene.

Rigid Body: A mass distribution in which the distance between any two points is constant.

Runge-Kutta Method: A numerical method to solve differential equations.

Simulated Annealing: An optimization technique based derived from the thermodynamic principles of annealing of metals.

Simulation: The process of subjecting mathematical models of system to mathematical models of effects to study outputs.

Superquadric: A solid that generalizes a quadric surface. The parametric representation of a superquadric is given by:

$$\begin{aligned} x &= a_1 \cos^{\epsilon_1}(\theta) \cos^{\epsilon_2}(\phi) \\ y &= a_2 \cos^{\epsilon_1}(\theta) \sin^{\epsilon_2}(\phi) \\ z &= a_3 \sin^{\epsilon_1}(\theta) \end{aligned}$$

Transformation Hierarchy: A tree consisting of modeling operations such as rotations, translations, scales, bends, twists, tapers, compositions, CSG operations, and primitives such as polygons and quadric surfaces. The primitives usually form the leaves of the tree and the internal nodes are associated with the transformations. The scope of any transformation is the subtree rooted at the node of the transformation. The composite transformation for any node is computed as the composite of the transformations encountered as the tree is traversed from the root to that node.

Scripts: A specification of the state of objects at different times in an animation.

Stiff System: A system with many widely different intrinsic time constants.

Tensor: A mathematical quantity that transforms under a tensor transformation rule when expressed in a different coordinate system.

Z-buffer: A hidden-surface elimination technique for rendering surfaces. A depth value is kept at each projected point on the screen. For each new point, if the new depth value is less than the stored depth value, the old value is overwritten.